Algorithms

AVL Tree

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

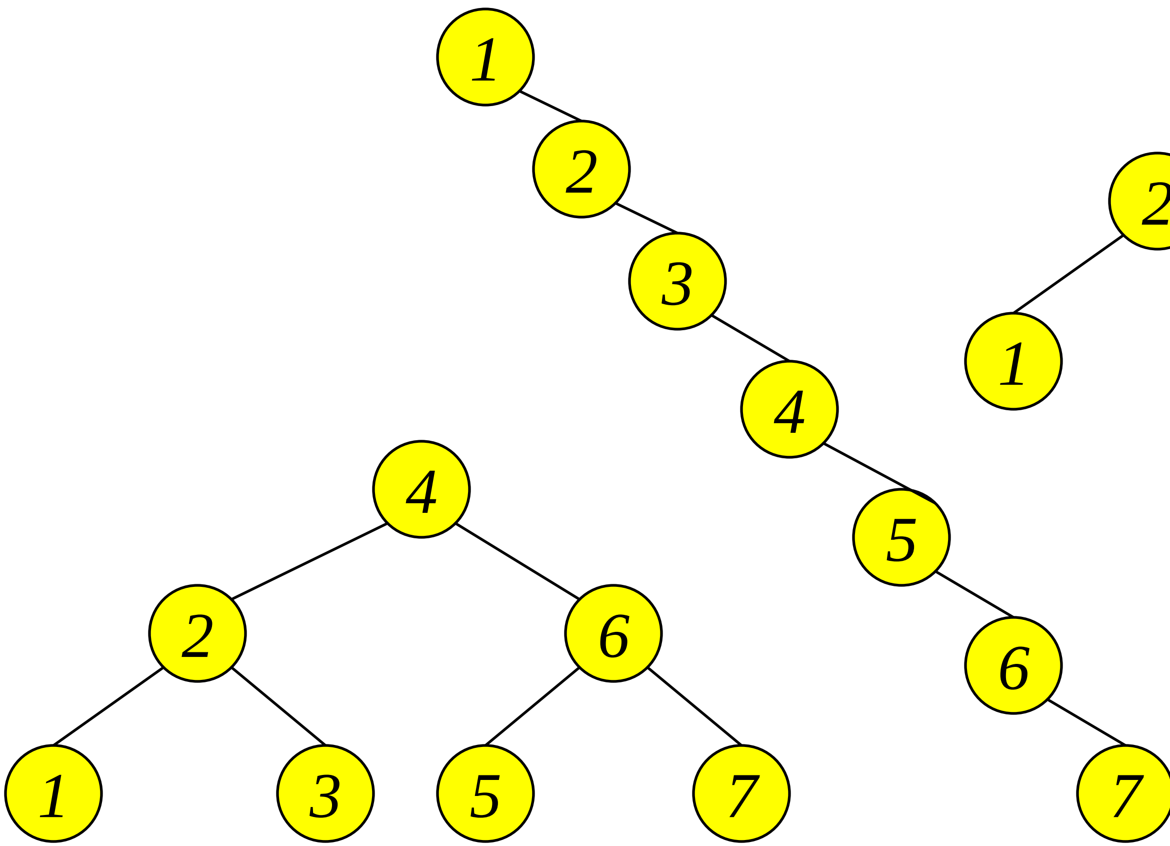
Binary Search Tree - Best Time

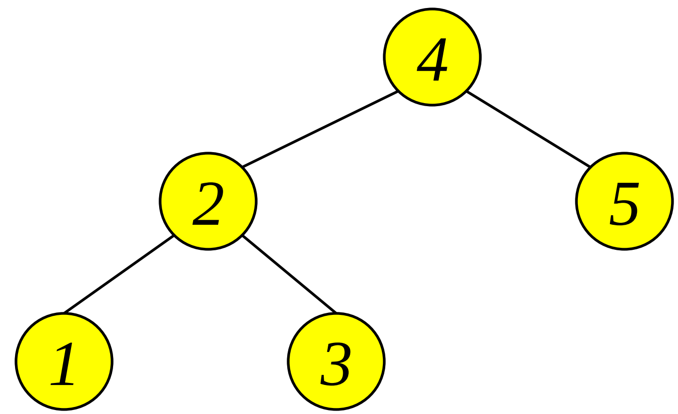
- All BST operations are O(h), where d is tree depth
- minimum d is $h = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree
  - Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?

Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

AVL Tree is…

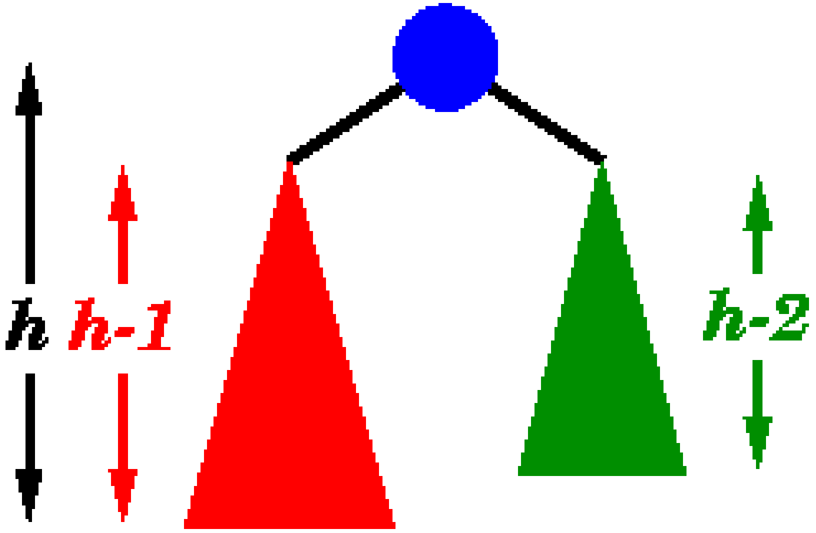
- Named after **A**delson-**V**elskii and **L**andis
- the first dynamically balanced trees to be propose
- Binary search tree with **balance condition** in which the sub-trees of each node can differ by at most 1 in their height

Definition of a balanced tree

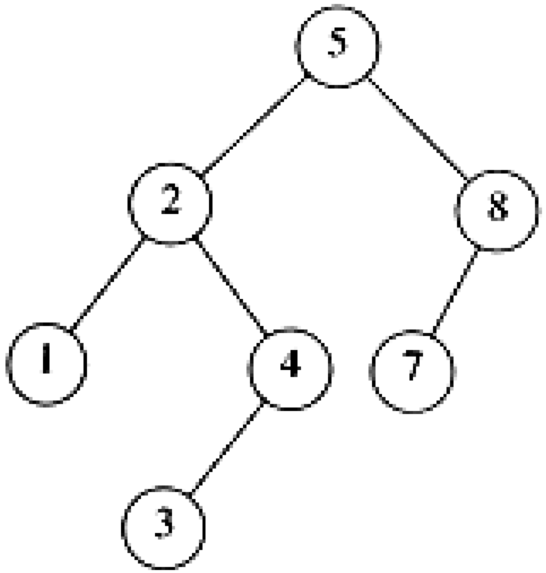
- Ensure the depth = O(log N)
- Take O(log N) time for searching, insertion, and deletion
- Every node must have left & right sub-trees of the same height

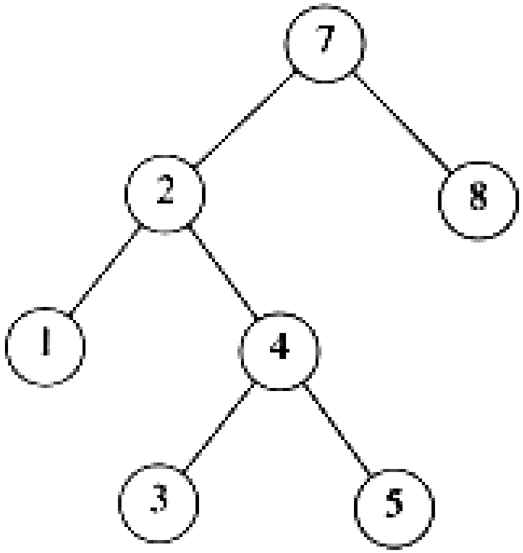
An AVL tree has the following properties:

1. Sub-trees of each node can differ by at most 1 in their height
2. Every sub-trees is an AVL tree



AVL tree?





YES

Each left sub-tree has height 1 greater than each right sub-tree

NO

Left sub-tree has height 3, but right sub-tree has height 1

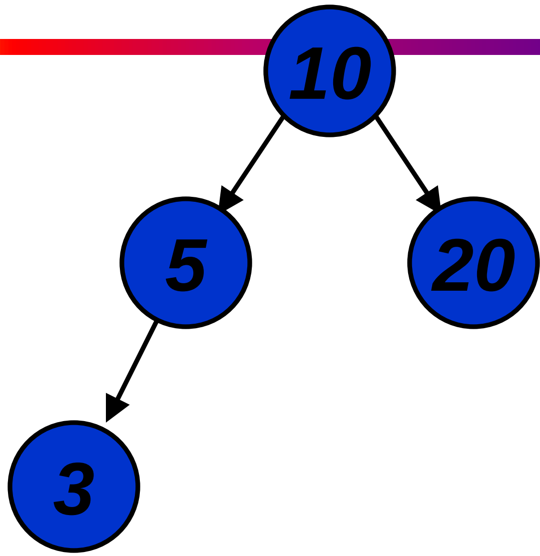
AVL tree

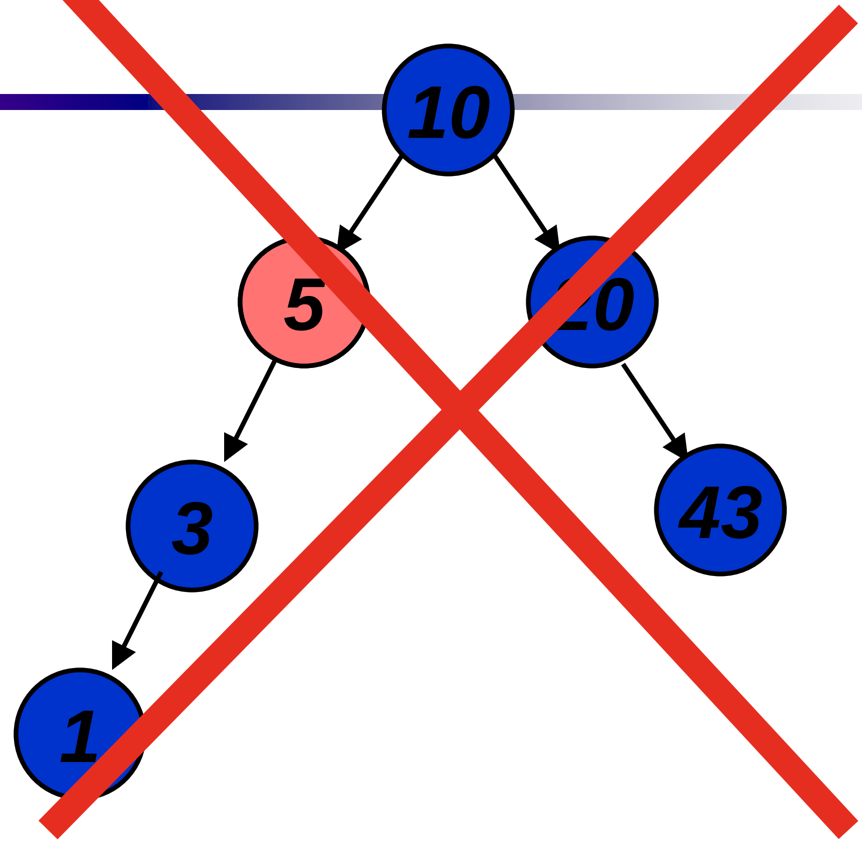
Height of a node

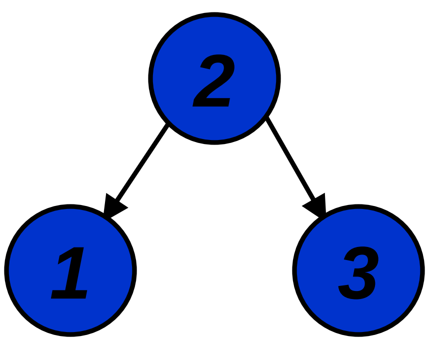
- The height of a leaf is 1. The height of a null pointer is zero.
- The height of an internal node is the maximum height of its children plus 1

Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

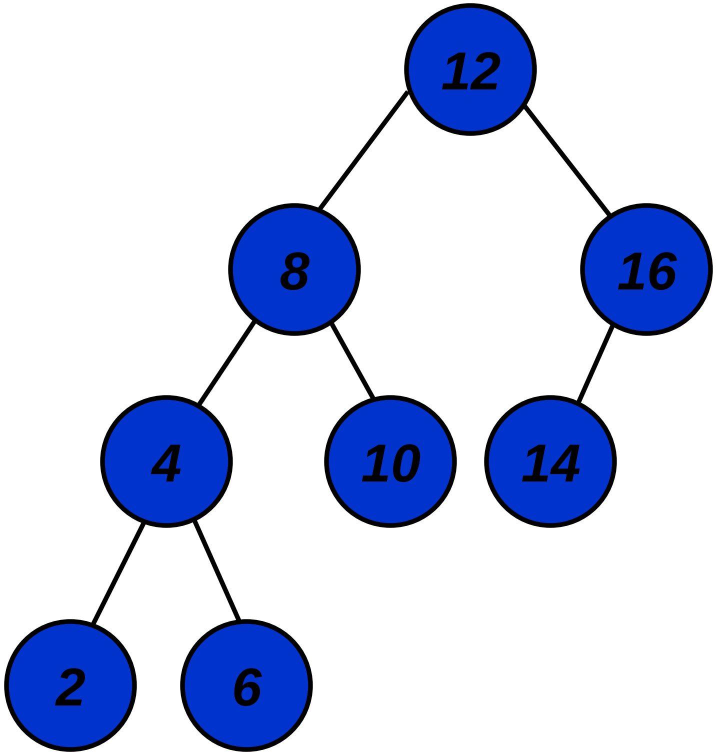
AVL Trees



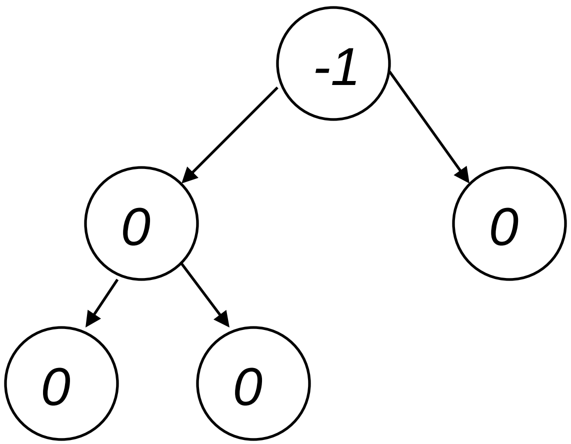




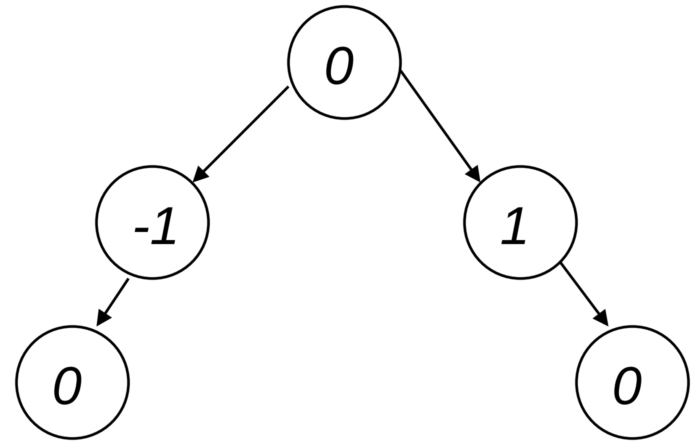
AVL Trees



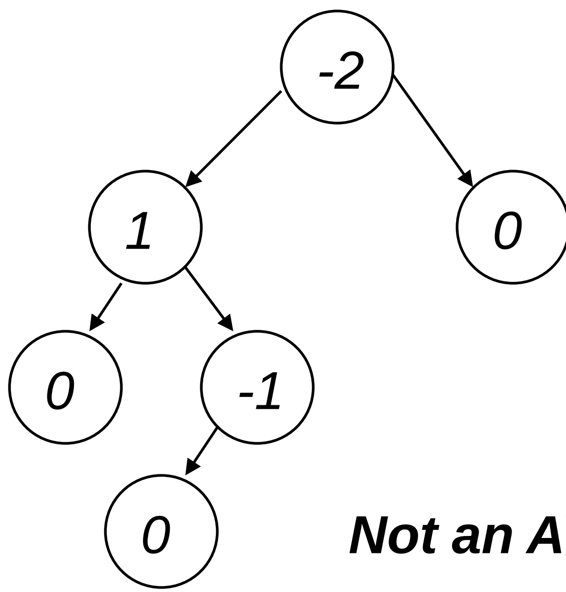
AVL Tree



AVL Tree

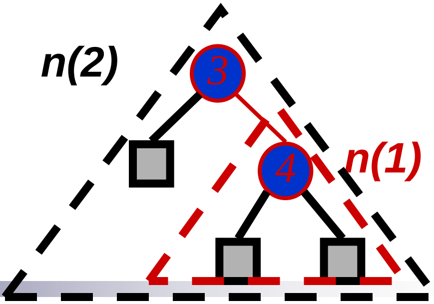


AVL Tree



Not an AVL Tree

Height of an AVL Tree



- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

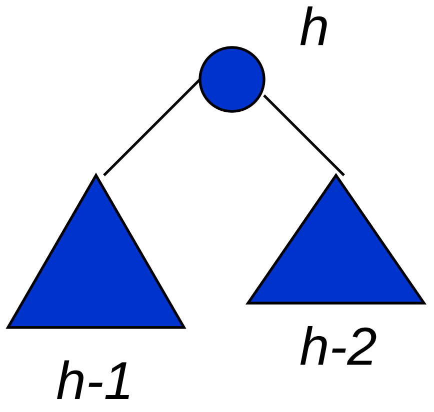
  $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

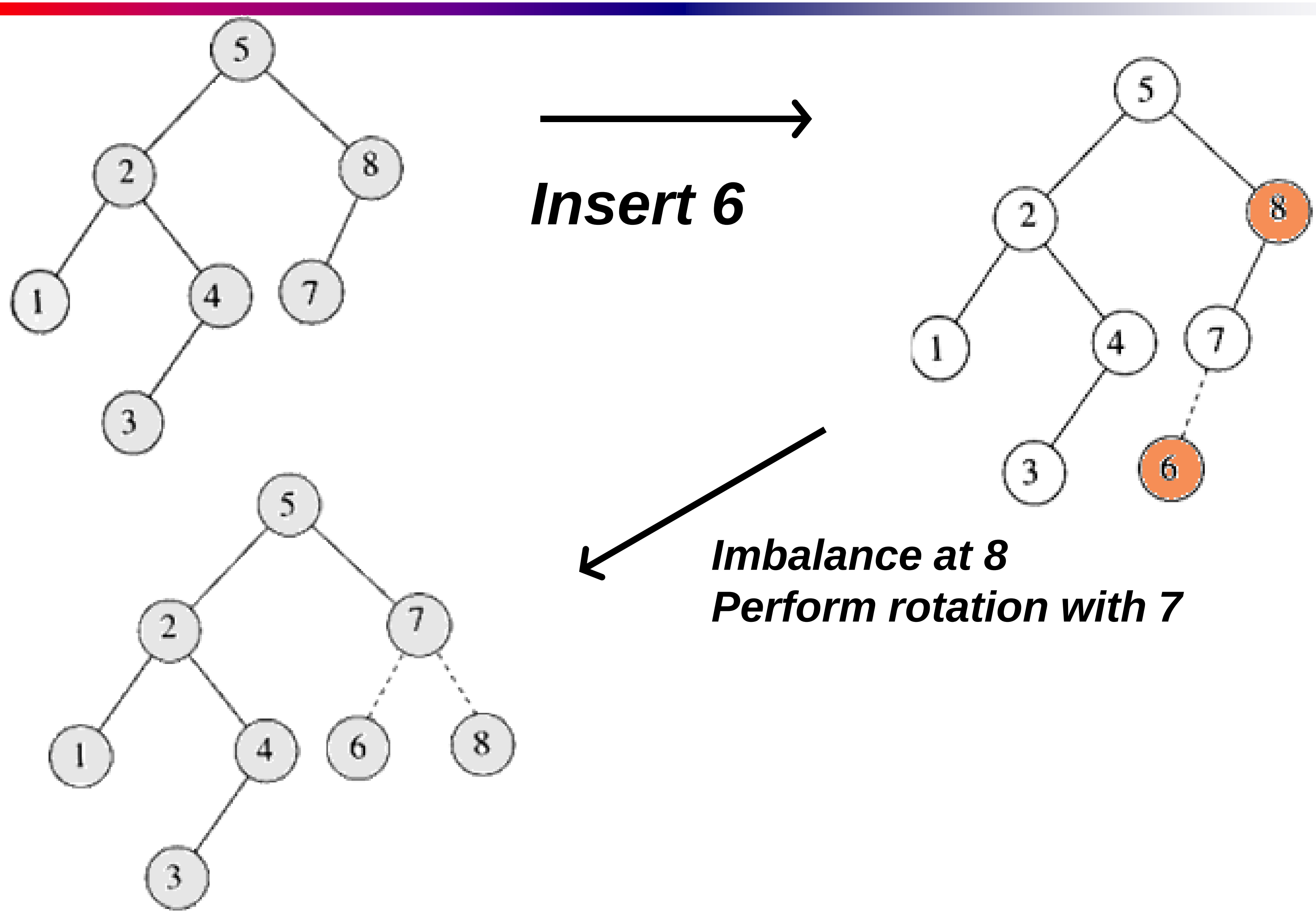
- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - N(0) = 1, N(1) = 2
- Induction
  - N(h) = N(h-1) + N(h-2) + 1
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



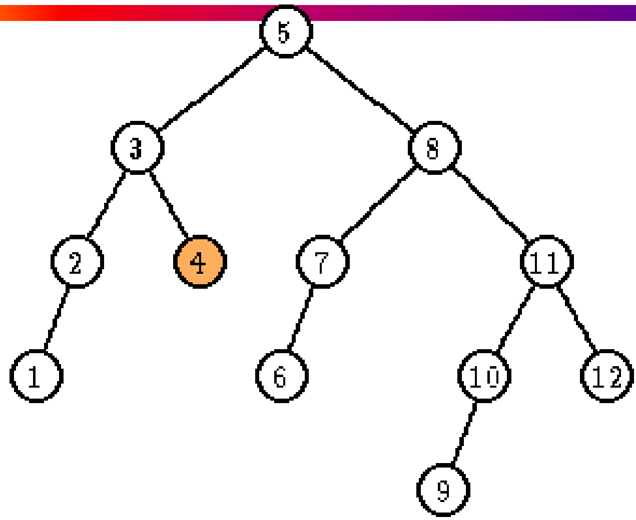
Height of an AVL Tree

- $N(h) \geq \phi^h$ $(\phi \approx 1.62)$
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(log n))

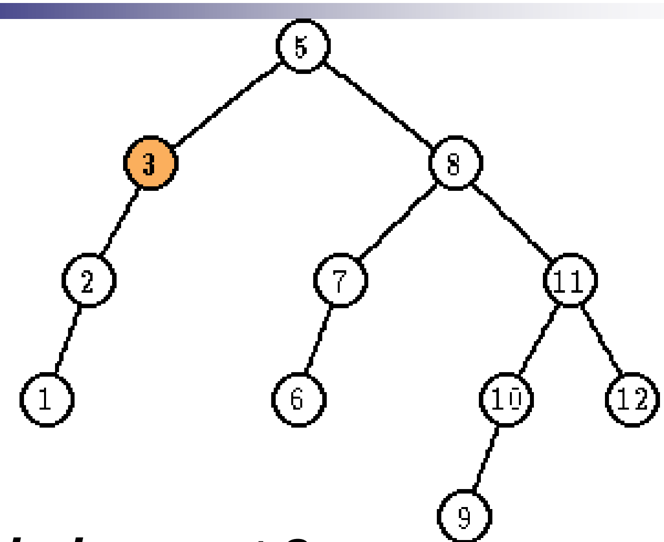
Insertion



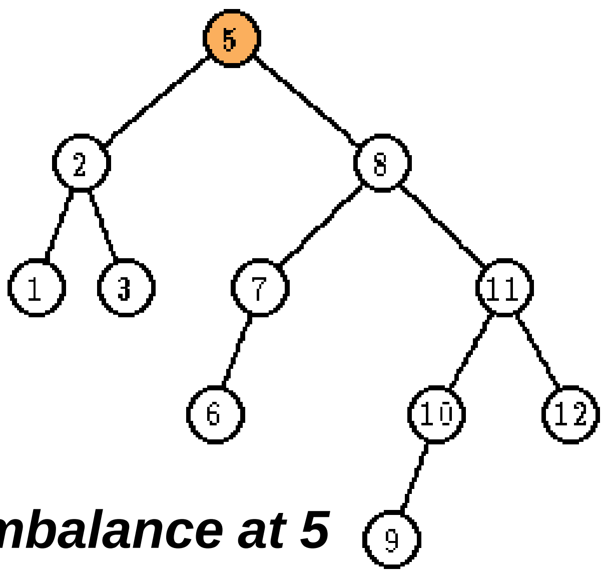
Deletion



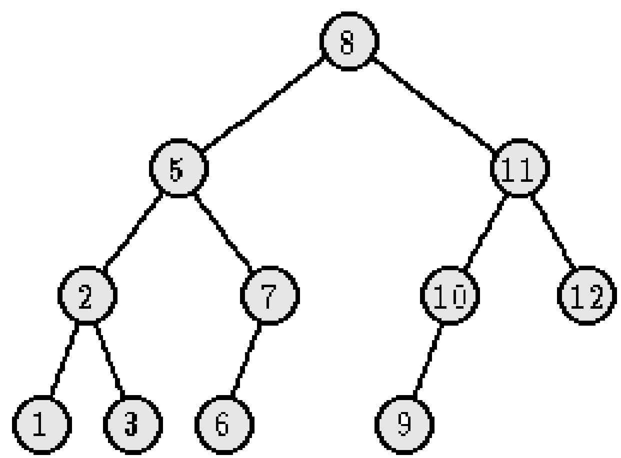
Delete 4



Imbalance at 3
Perform rotation with 2



Imbalance at 5
Perform rotation with 8



Key Points

- AVL tree remain balanced by applying rotations, therefore it guarantees O(log N) search time in a dynamic environment
- Tree can be re-balanced in at most O(log N) time

Searching AVL Trees

- Searching an AVL tree is exactly the same as searching a regular binary tree
  - all descendants to the right of a node are greater than the node
  - all descendants to the left of a node are less than the node

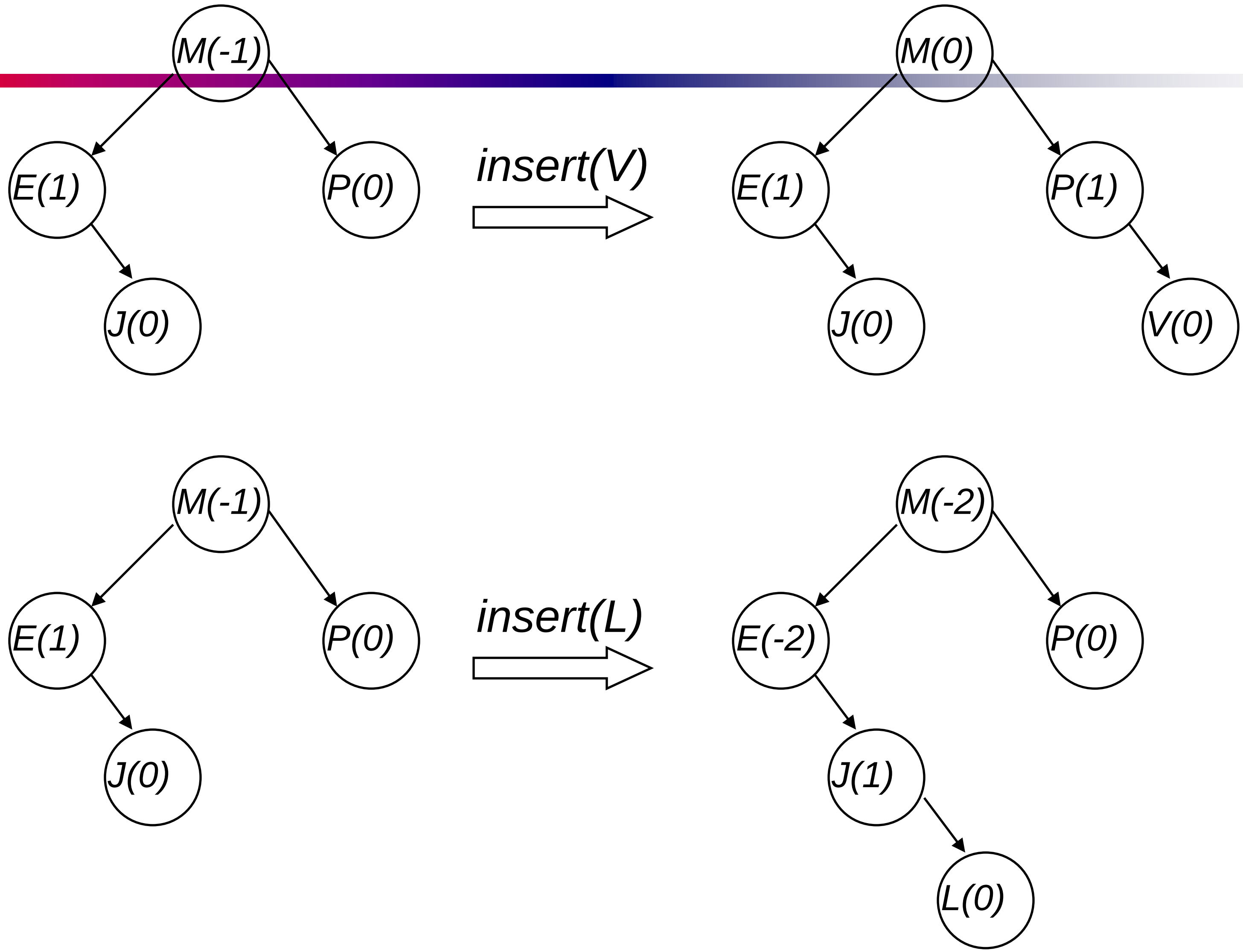
Inserting in AVL Tree

- Insertion is similar to regular binary tree
  - keep going left (or right) in the tree until a null child is reached
  - insert a new node in this position
    - an inserted node is *always* a leaf to start with
- Major difference from binary tree
  - must check if any of the sub-trees in the tree have become too unbalanced
    - search from inserted node to root looking for any node with a balance factor of 2

Inserting in AVL Tree

- A few points about tree inserts
  - the insert will be done recursively
  - the insert call will return true if the height of the sub-tree has changed
    - since we are doing an insert, the height of the sub-tree can only increase
  - if *insert()* returns true, balance factor of current node needs to be adjusted
    - balance factor = height(right) – height(left)
      - left sub-tree increases, balance factor decreases by 1
      - right sub-tree increases, balance factor increases by 1
  - if balance factor equals   2 for any node, the sub-tree must be rebalanced

Inserting in AVL Tree



This tree needs to be fixed!

Re-Balancing a Tree

- To check if a tree needs to be rebalanced
  - start at the parent of the inserted node and journey up the tree to the root
    - if a node's balance factor becomes   2 need to do a rotation in the sub-tree rooted at the node
    - once sub-tree has been re-balanced, guaranteed that the rest of the tree is balanced as well
      - can just return false from the *insert()* method
  - 4 possible cases for re-balancing
    - only 2 of them need to be considered
      - other 2 are identical but in the opposite direction

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. *Insertion into left subtree of left child of α.*
2. *Insertion into right subtree of right child of α.*

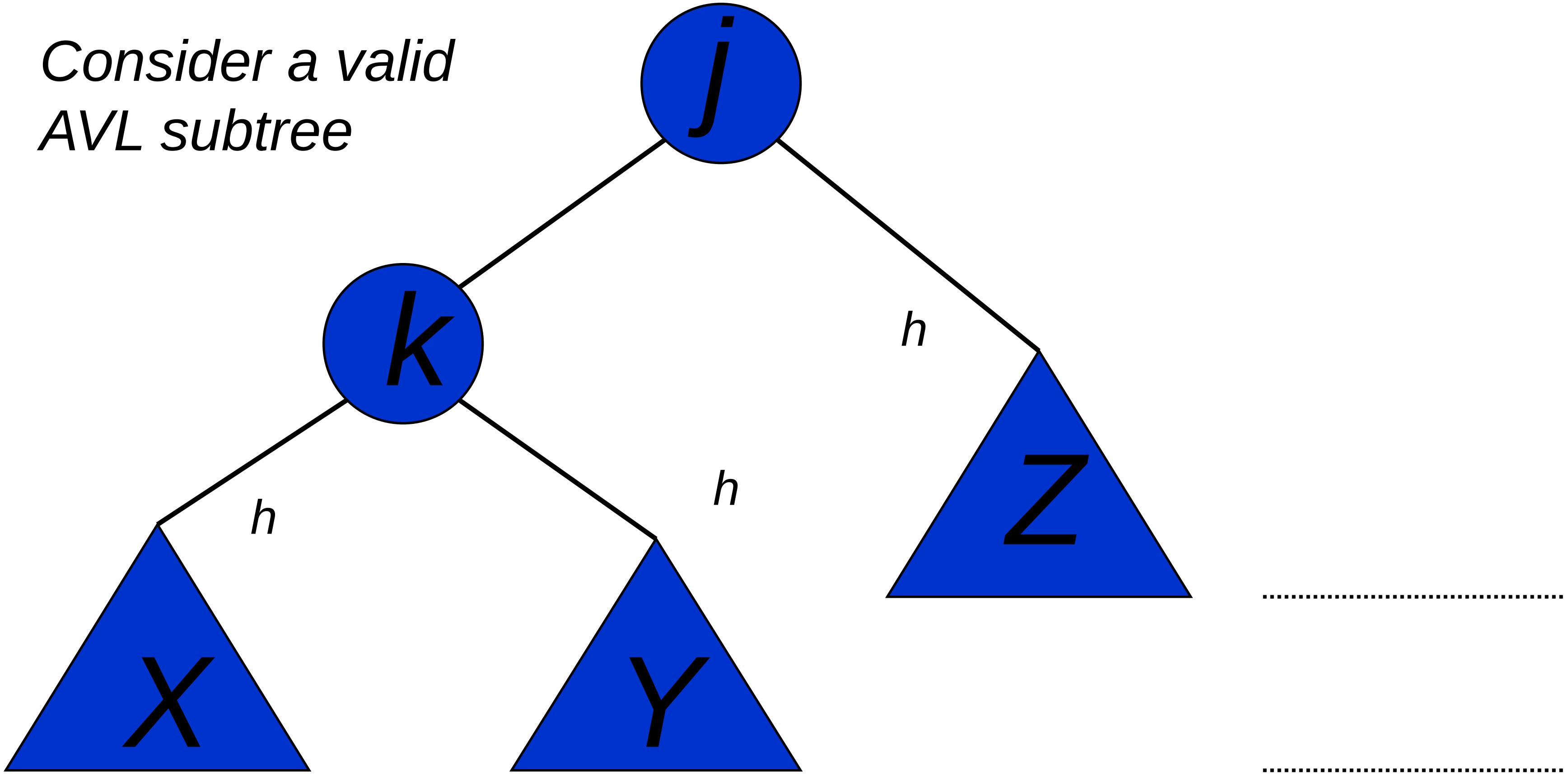
Inside Cases (require double rotation) :

3. *Insertion into right subtree of left child of α.*
4. *Insertion into left subtree of right child of α.*

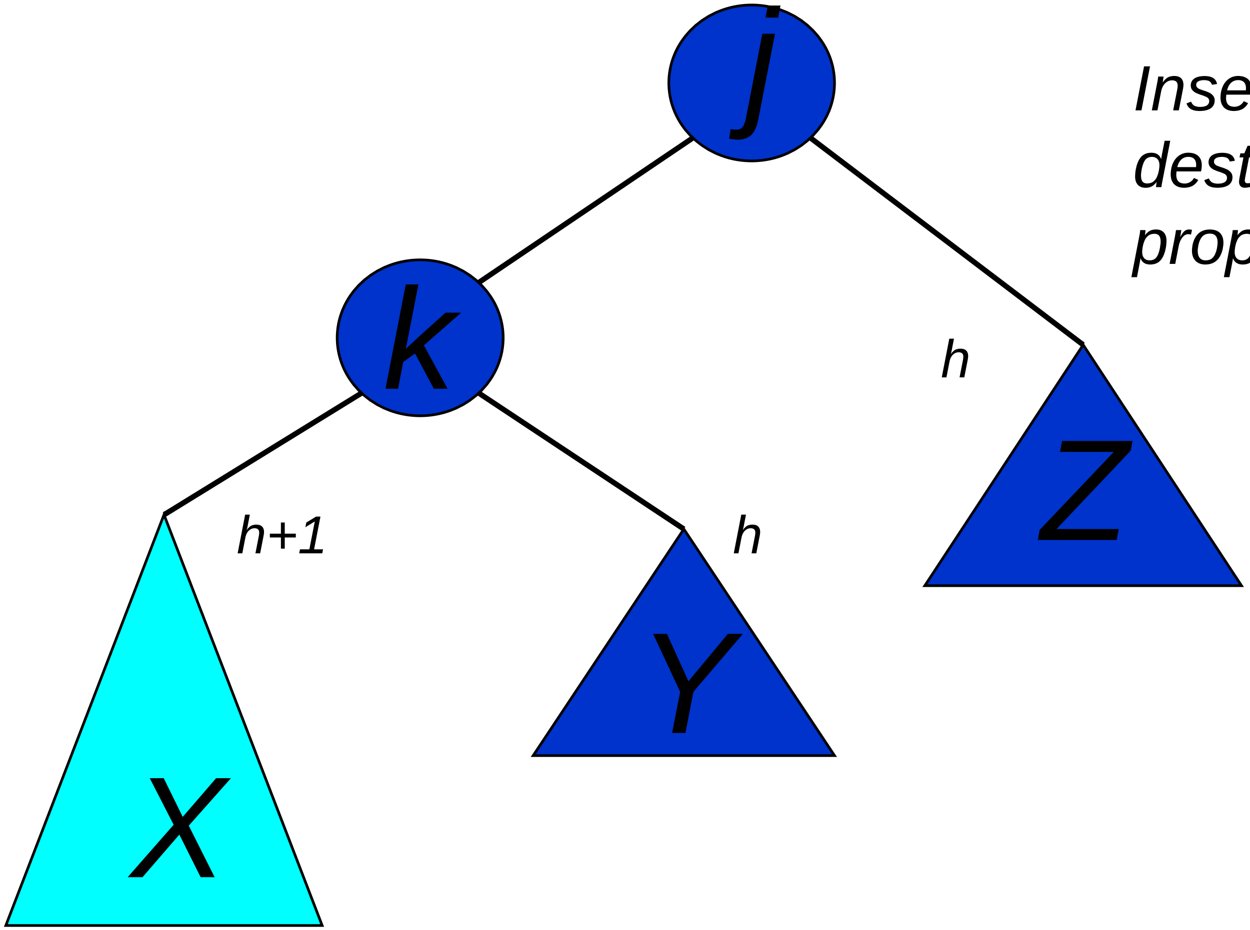
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid AVL subtree

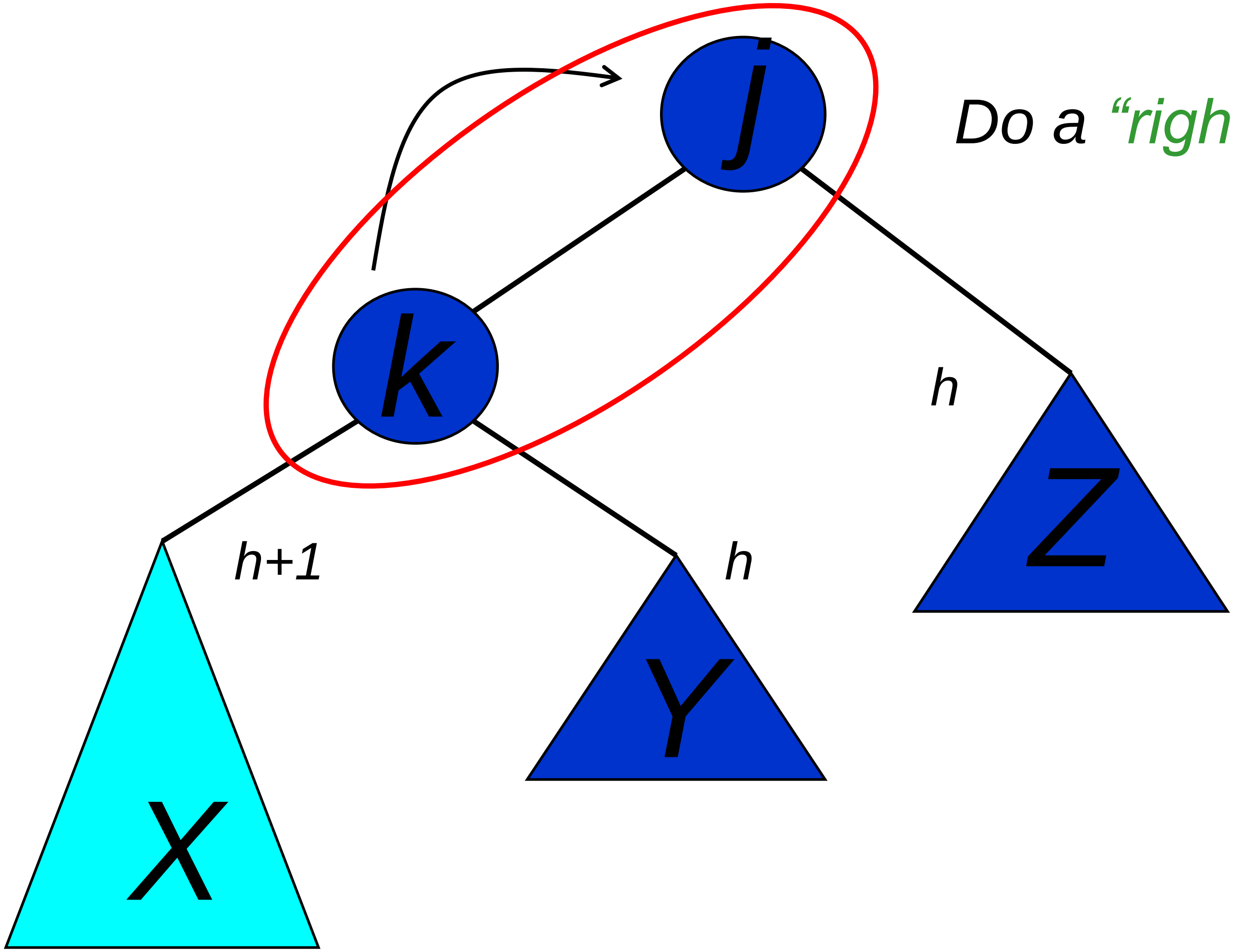


AVL Insertion: Outside Case



Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



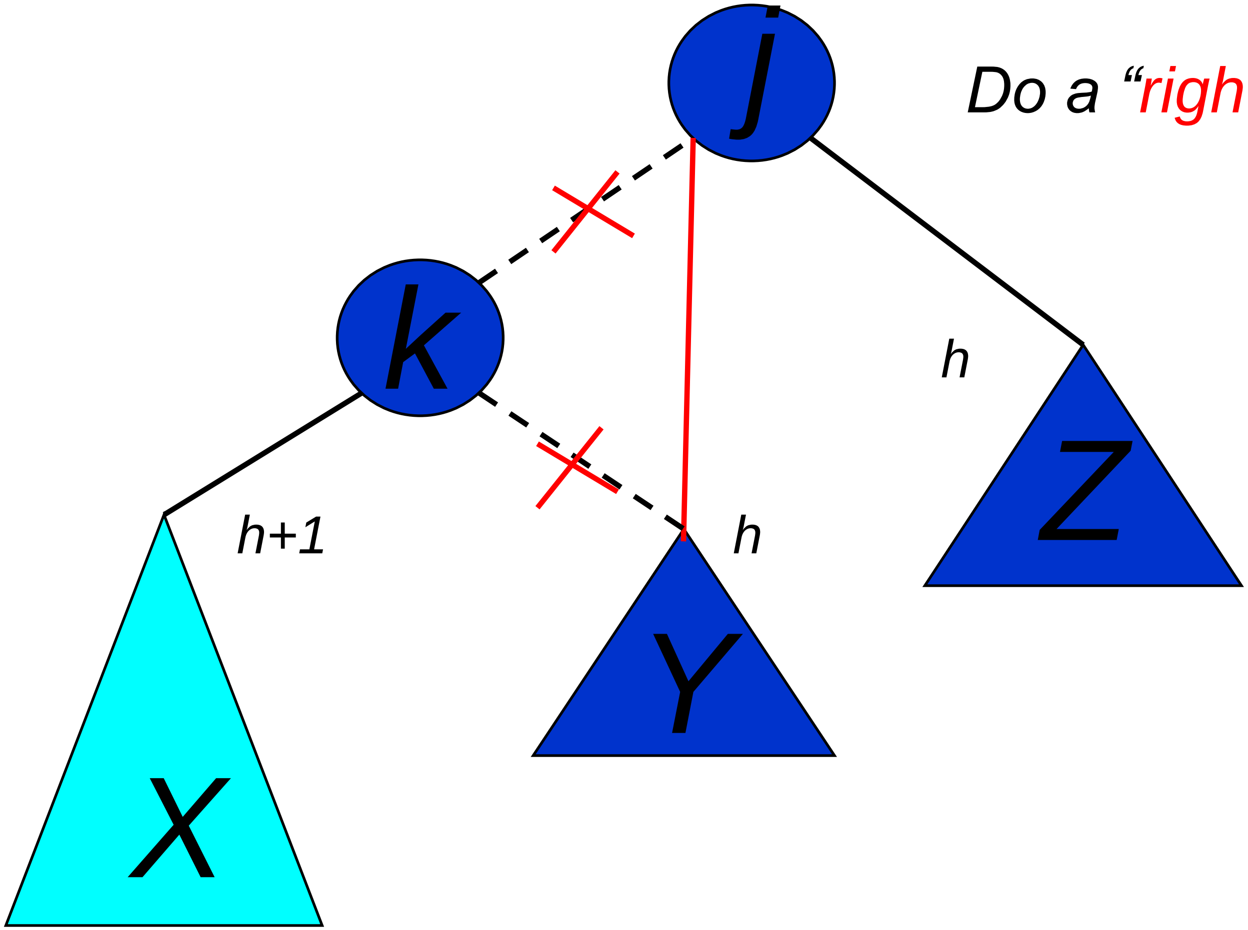
Do a *"right rotation"*

....

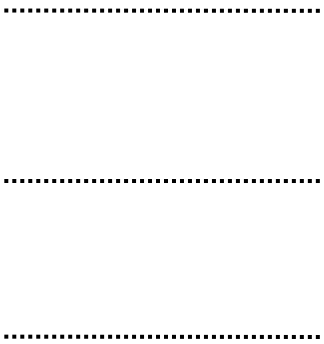
....

....

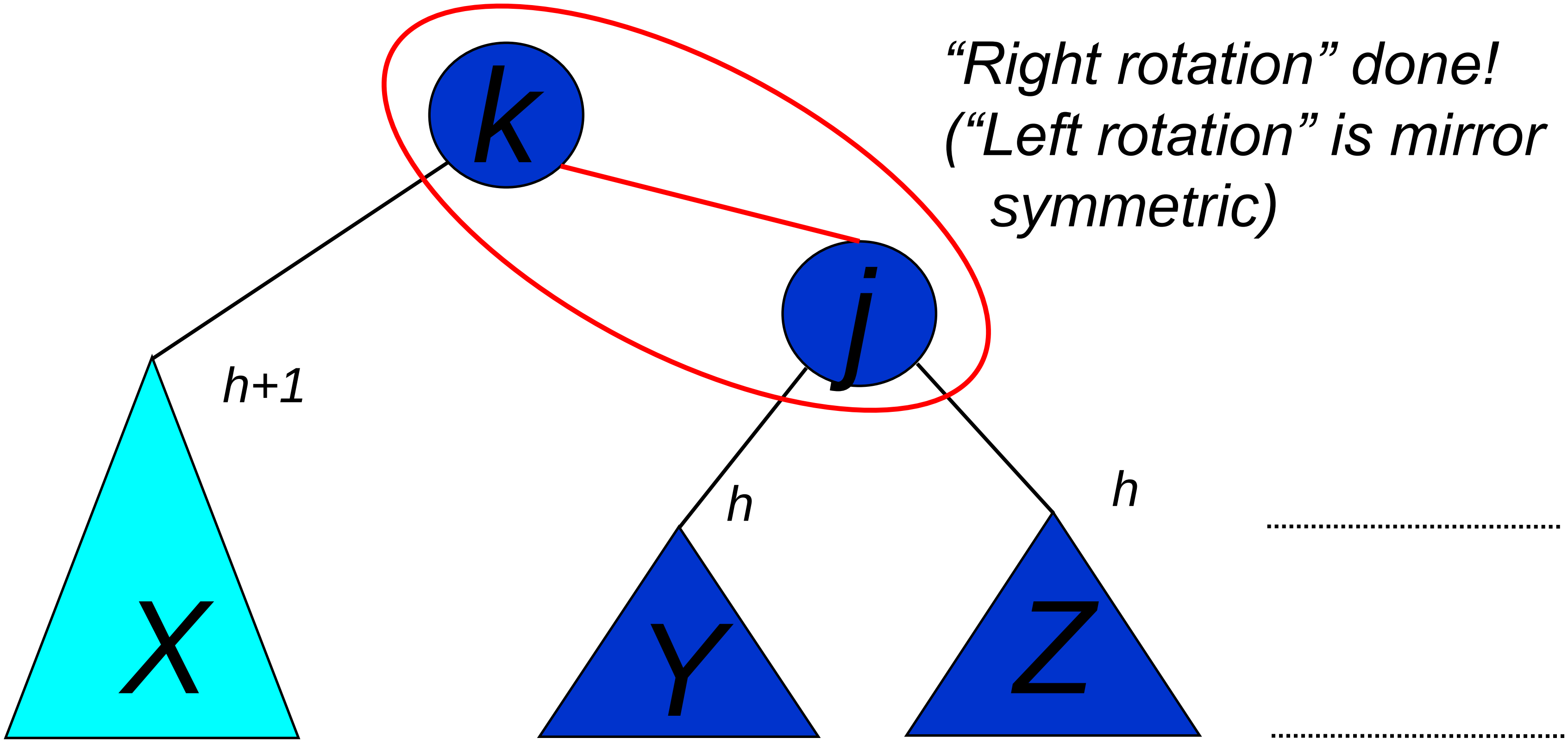
Single right rotation



Do a "right rotation"



Outside Case Completed

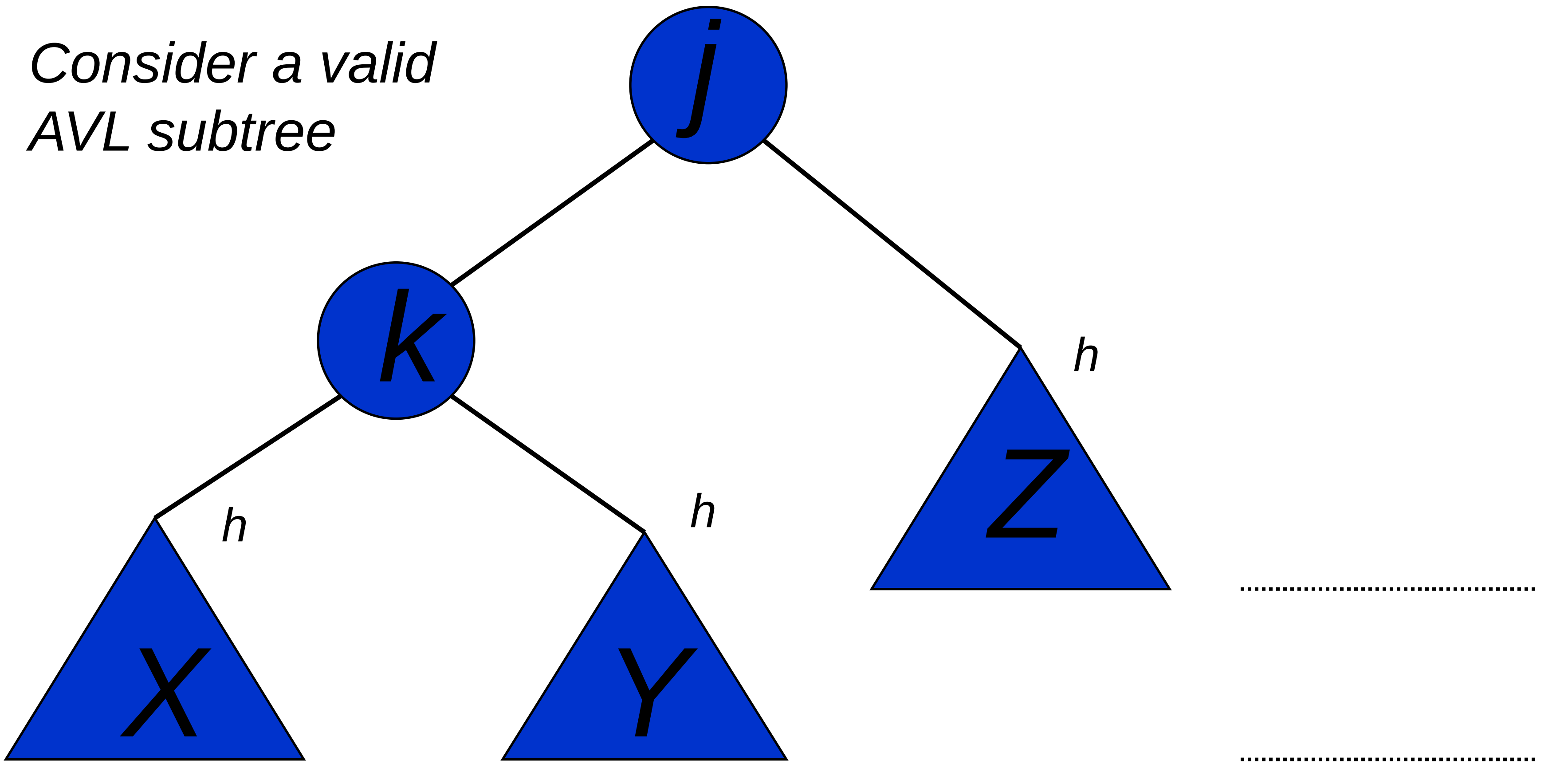


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

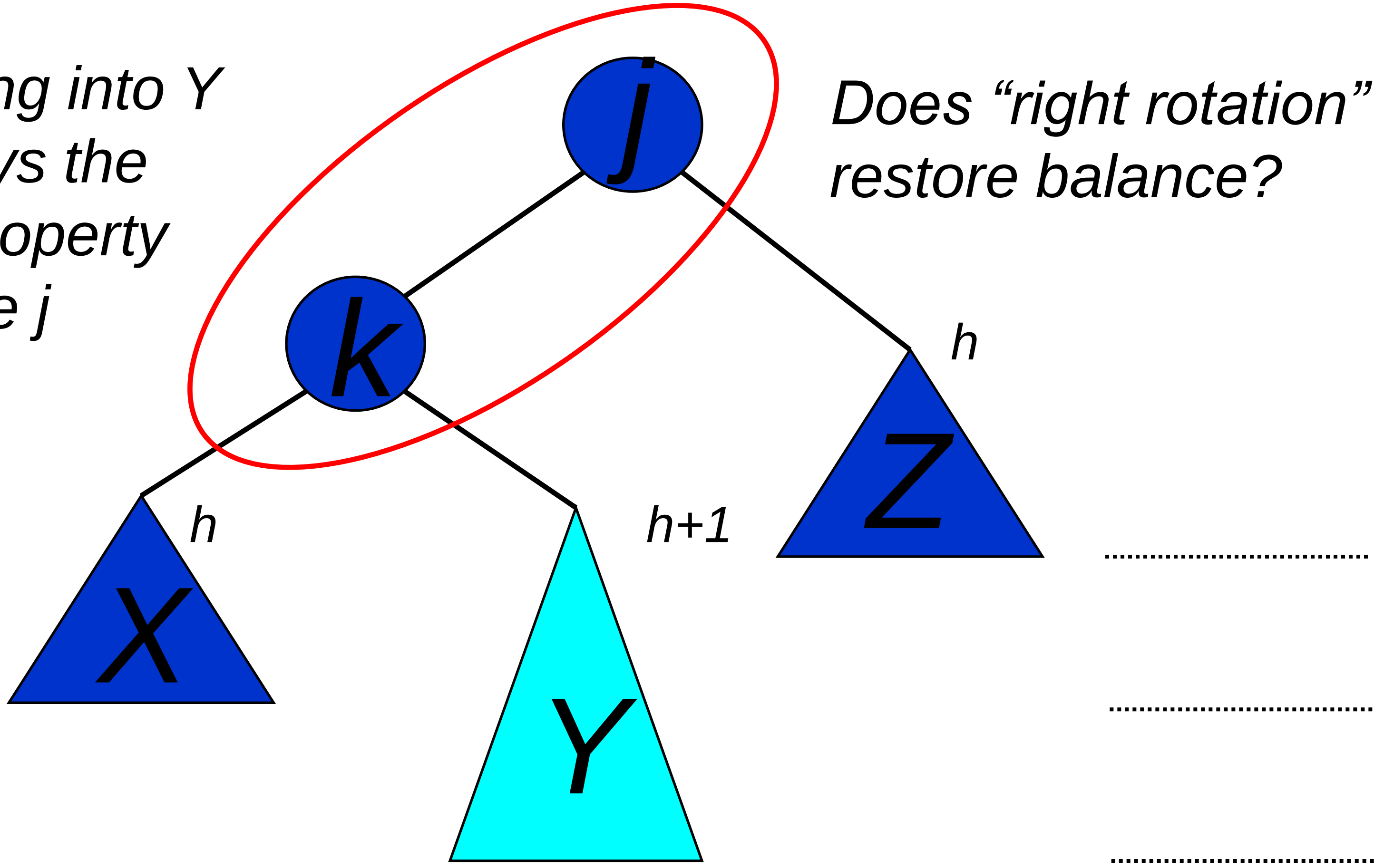
AVL Insertion: Inside Case

Consider a valid AVL subtree



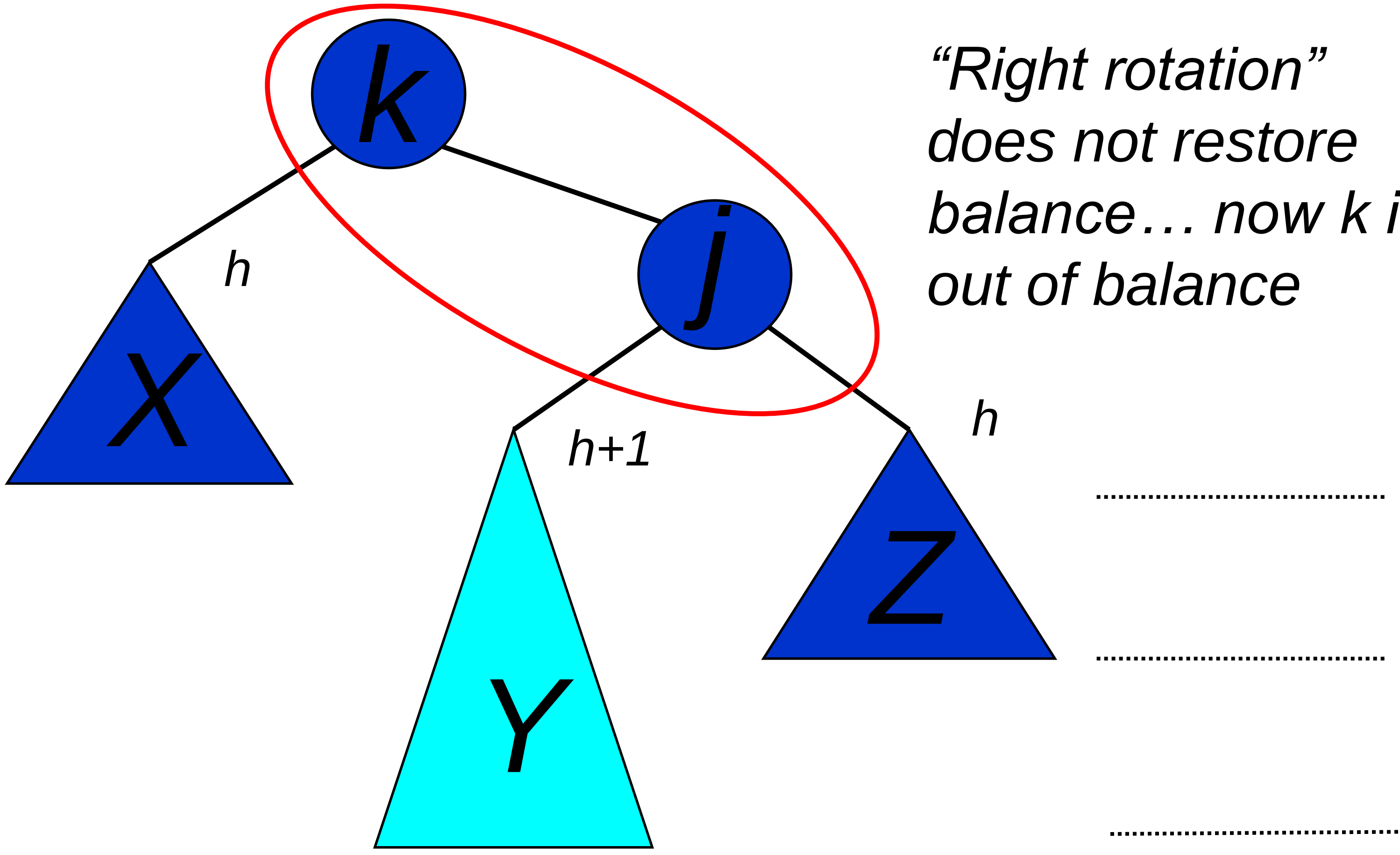
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "right rotation" restore balance?

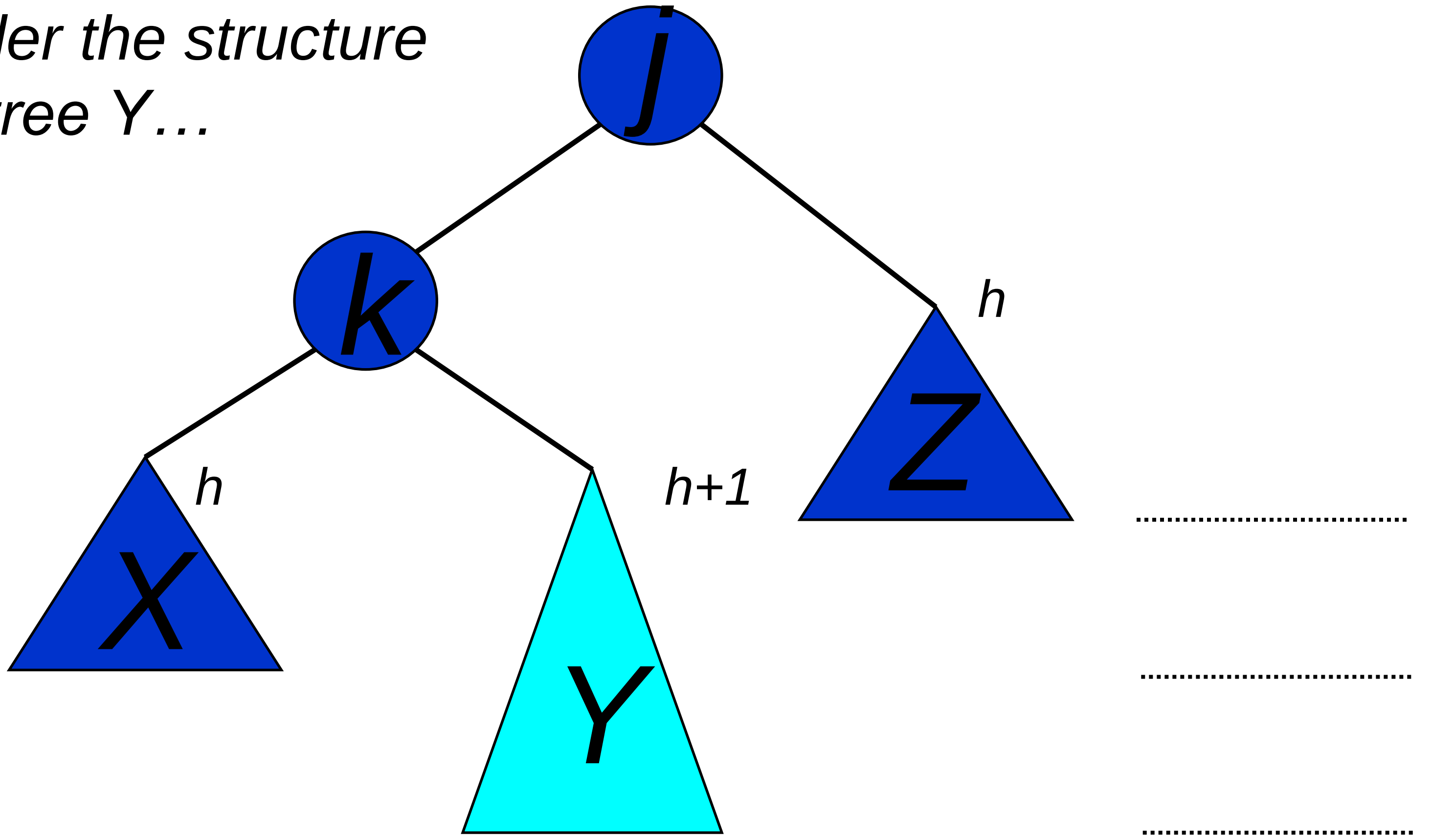
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

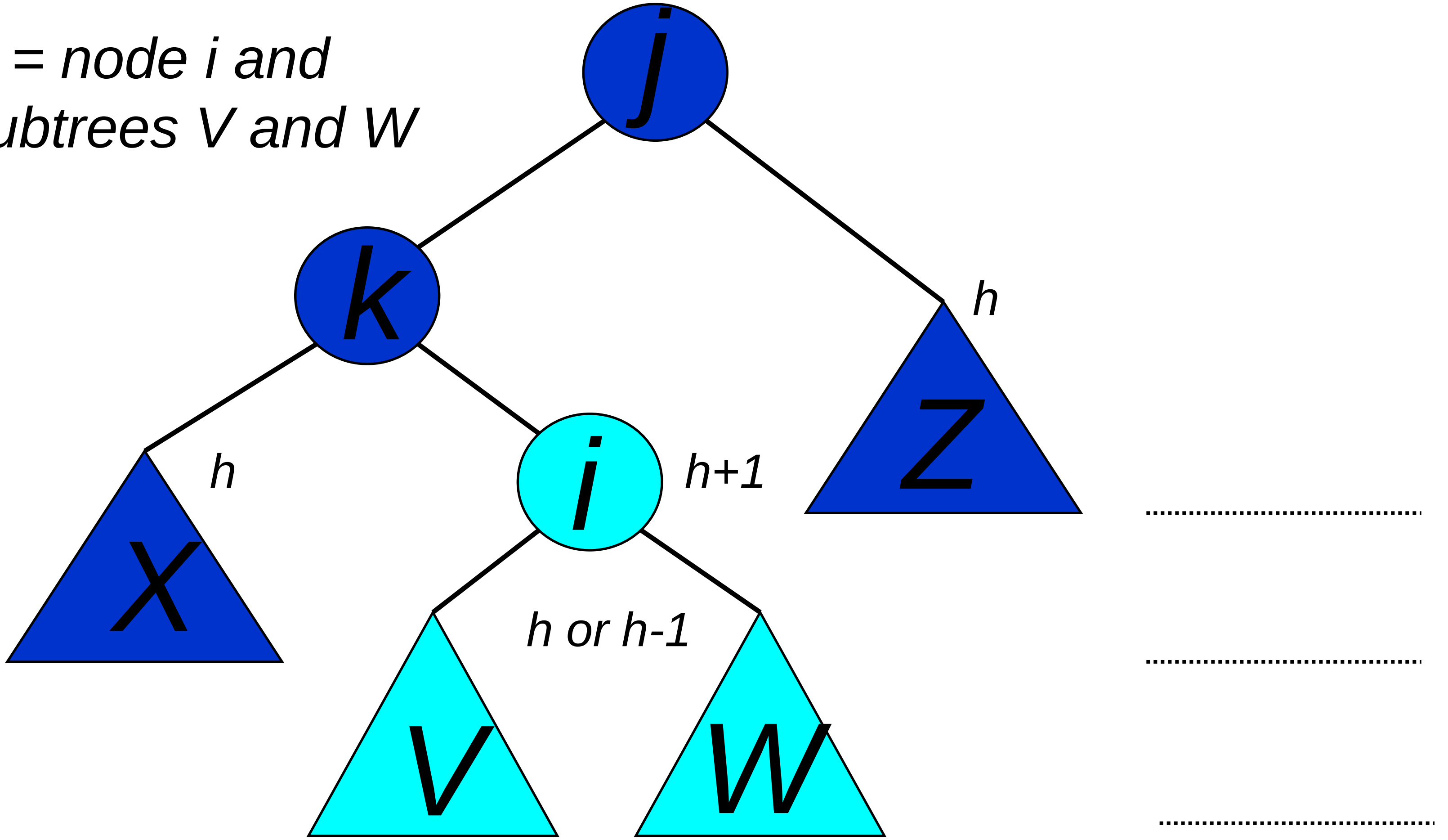
AVL Insertion: Inside Case

Consider the structure of subtree Y…

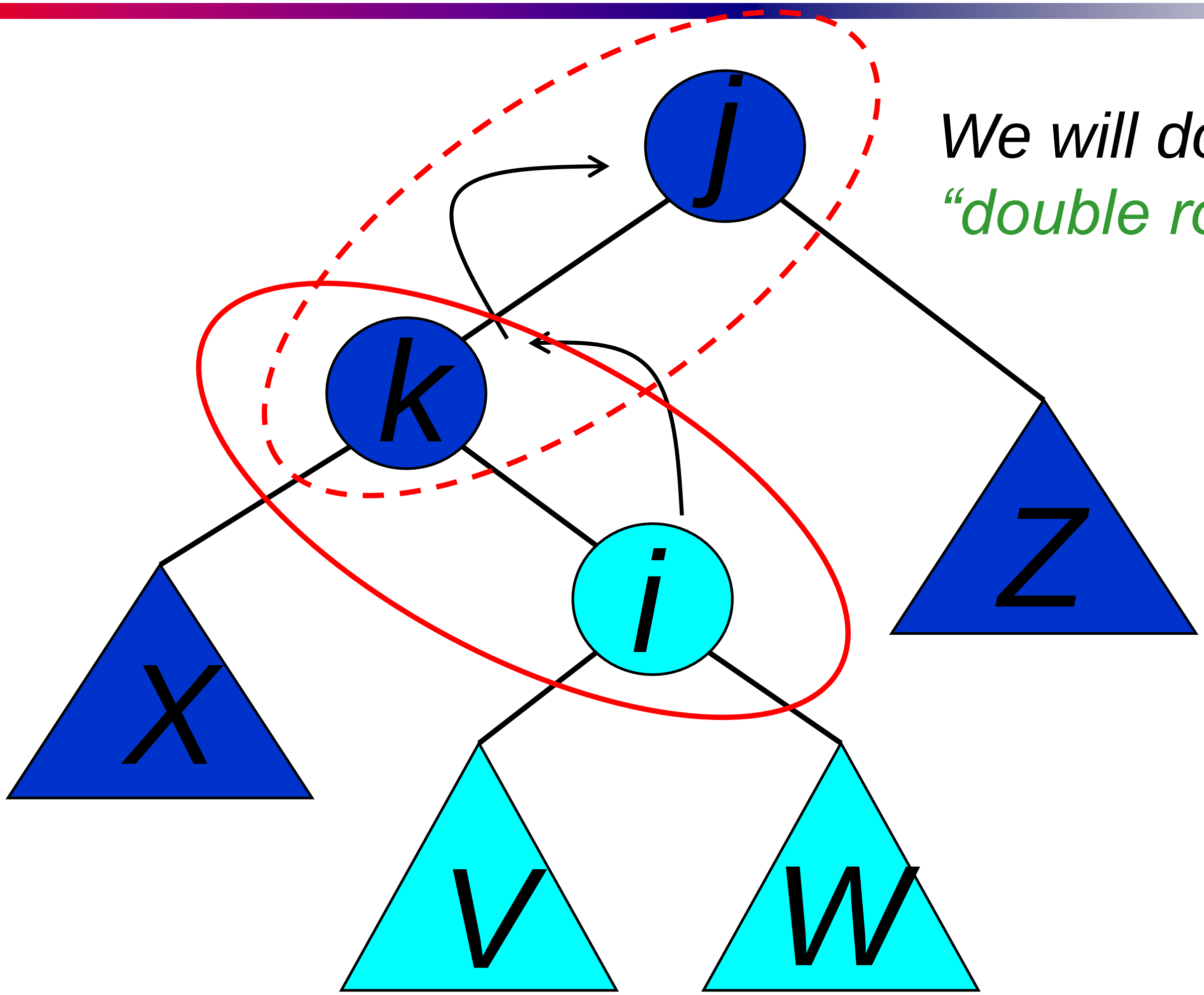


AVL Insertion: Inside Case

Y = node i and subtrees V and W



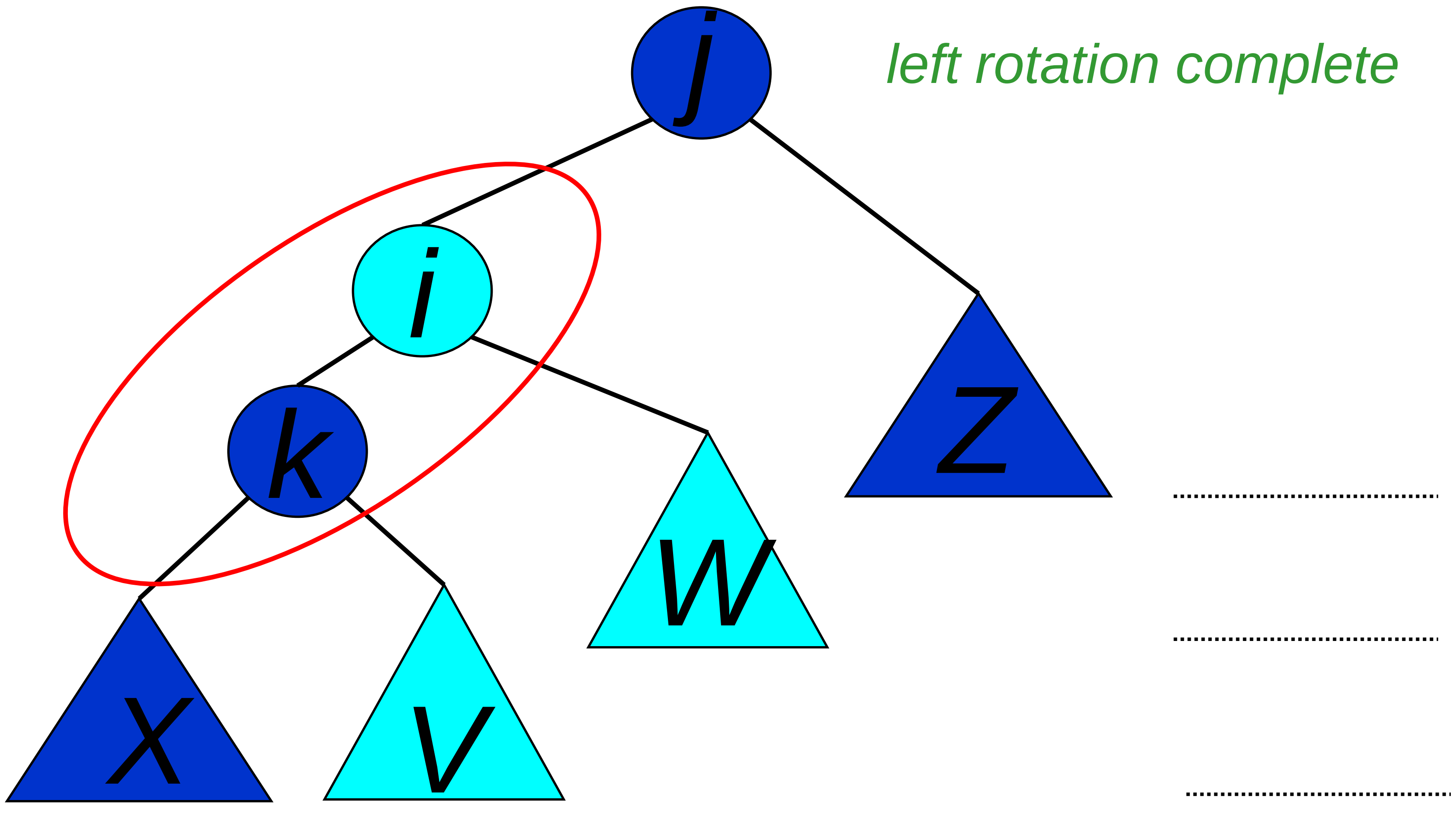
AVL Insertion: Inside Case



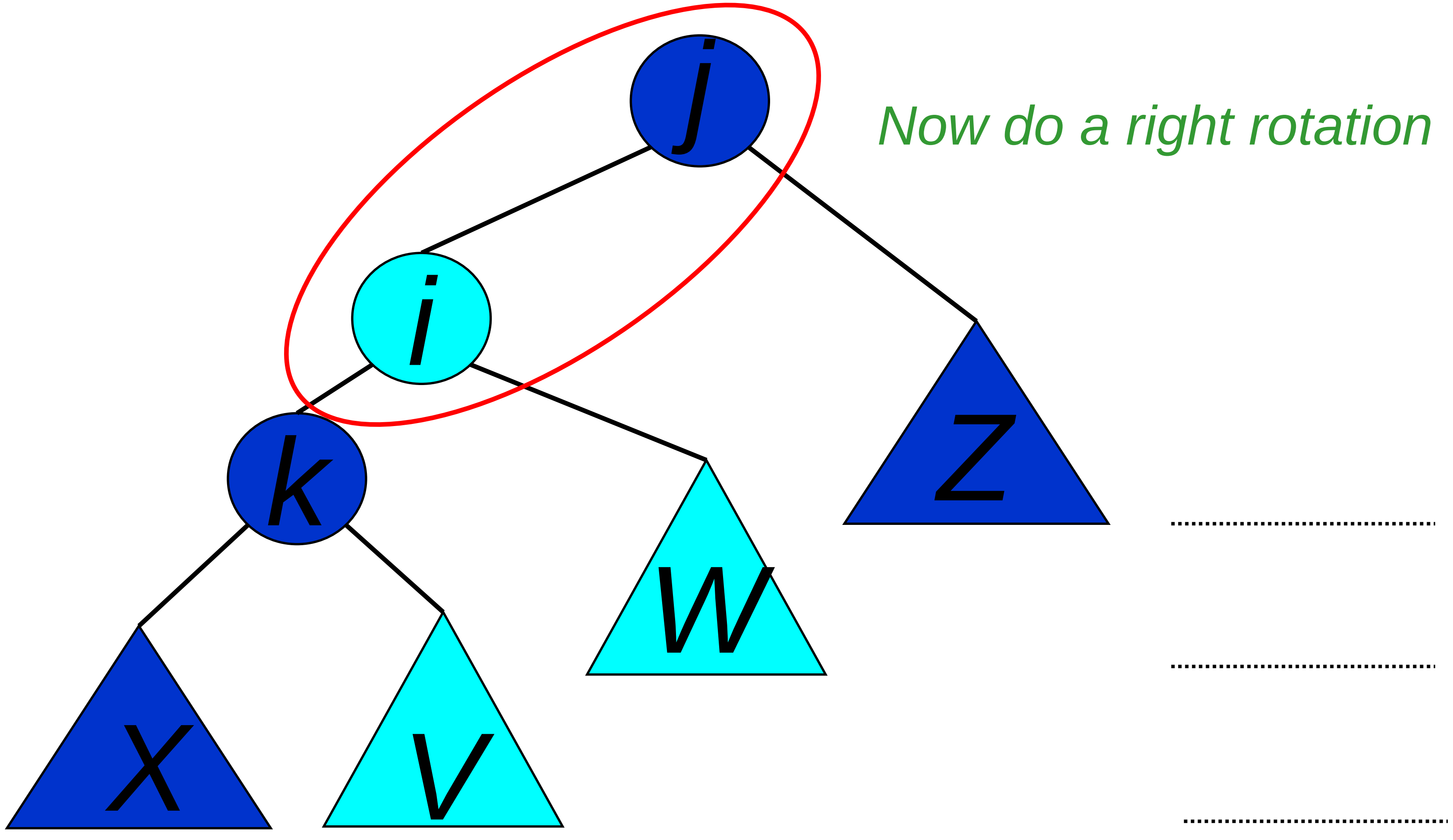
We will do a left-right "double rotation" . . .

Double rotation : first rotation

left rotation complete



Double rotation : second rotation

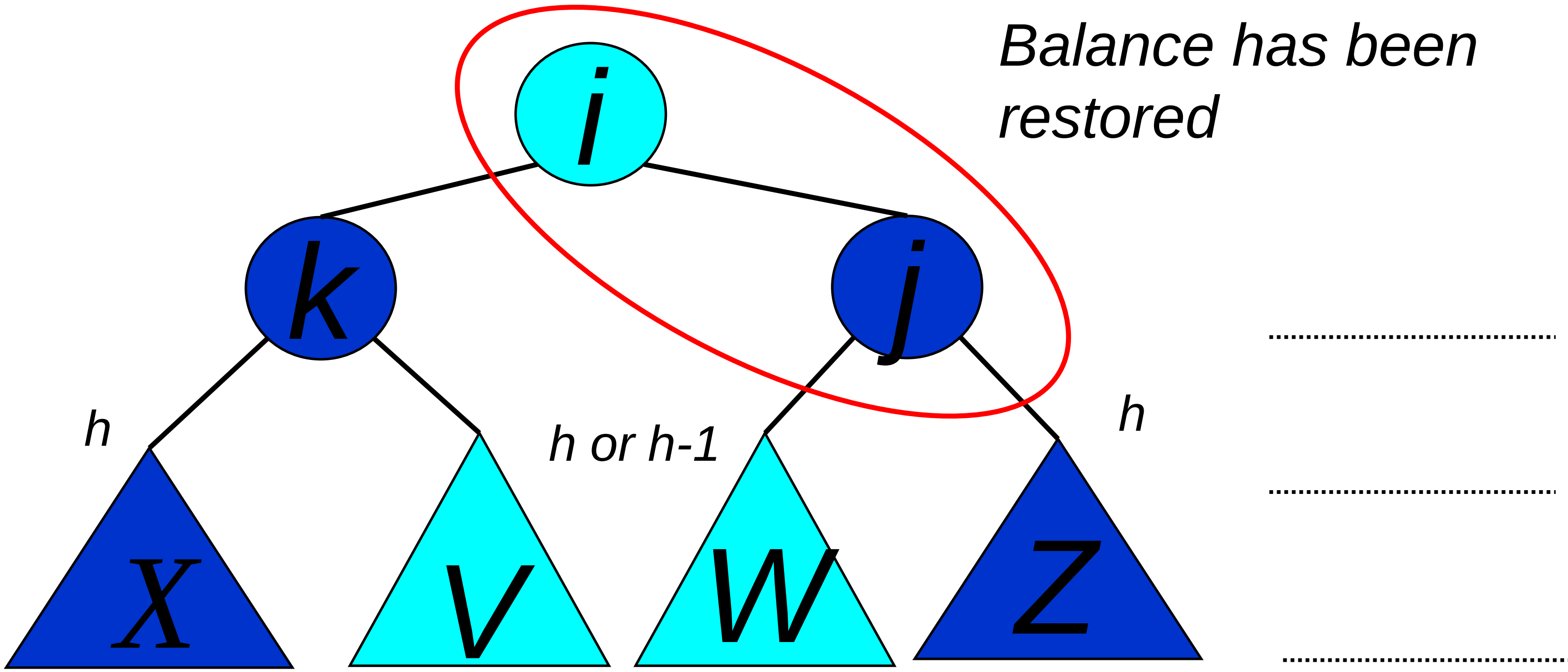


Now do a right rotation

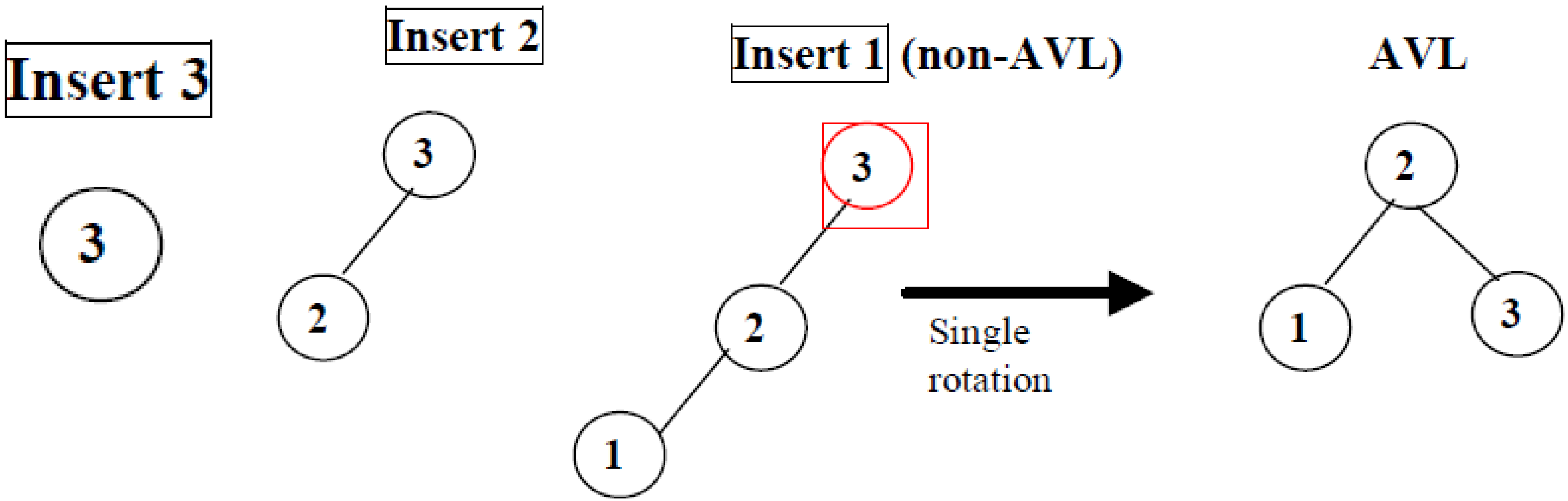
Double rotation : second rotation

right rotation complete

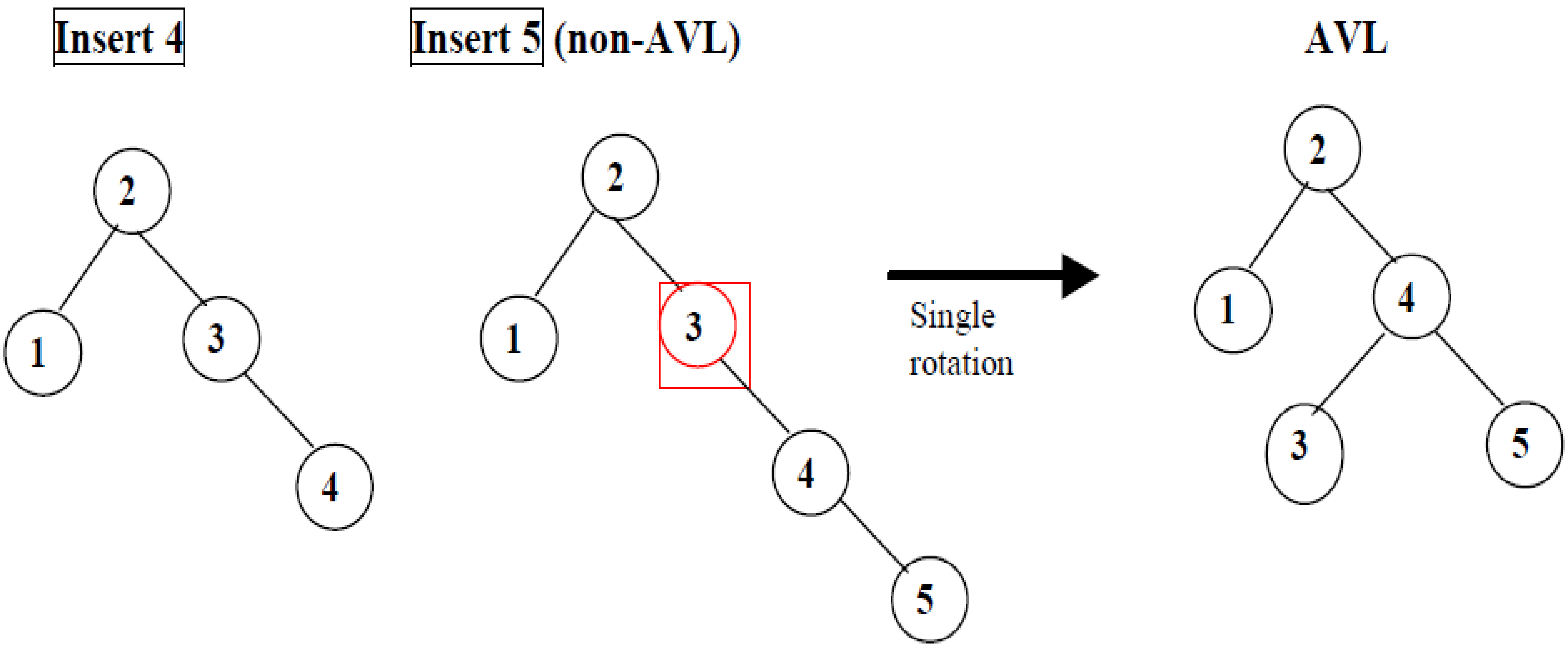
Balance has been restored



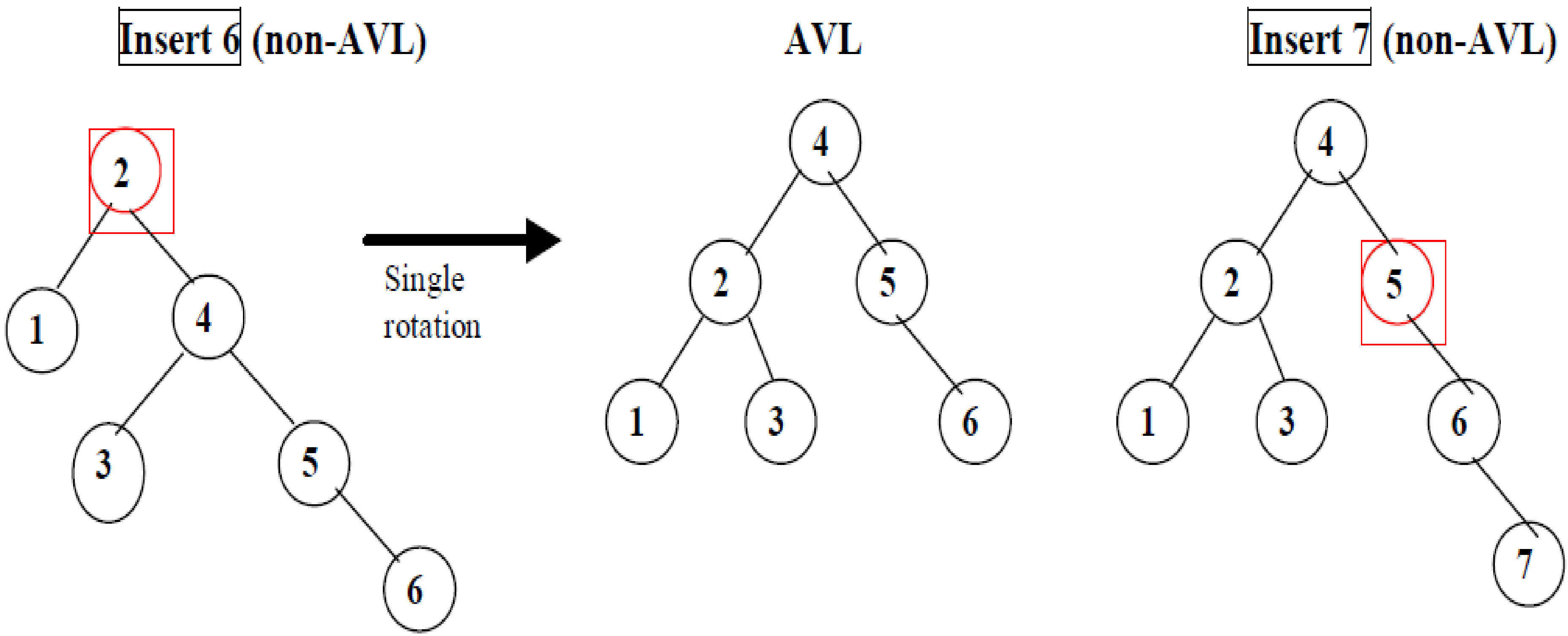
AVL Trees Example



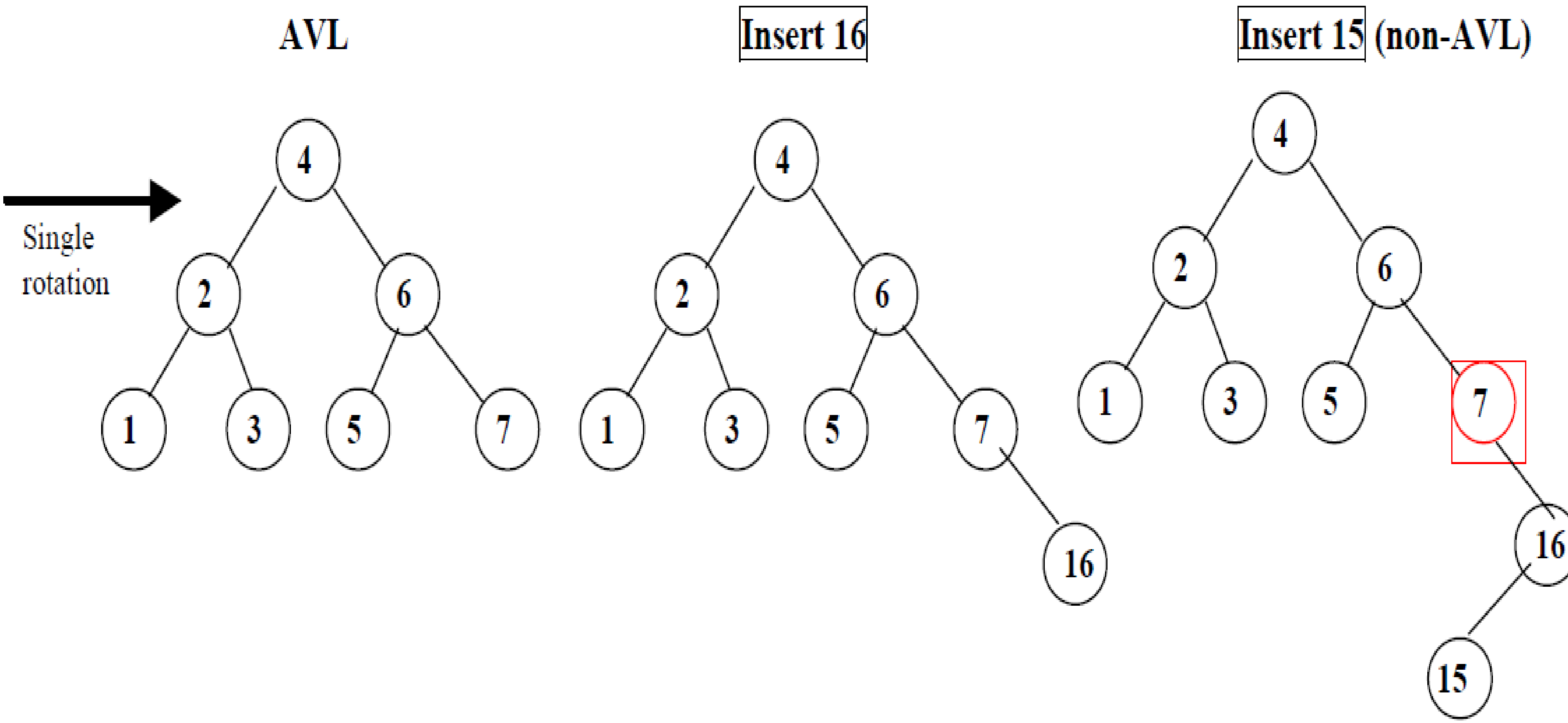
AVL Trees Example



AVL Trees Example

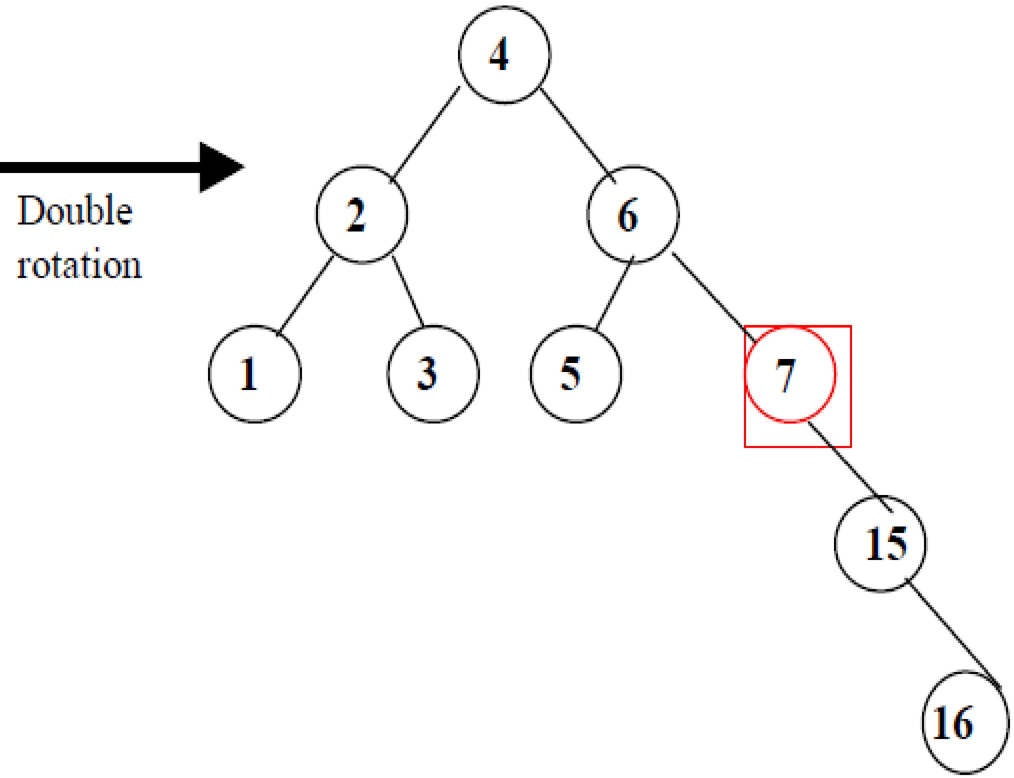


AVL Trees Example

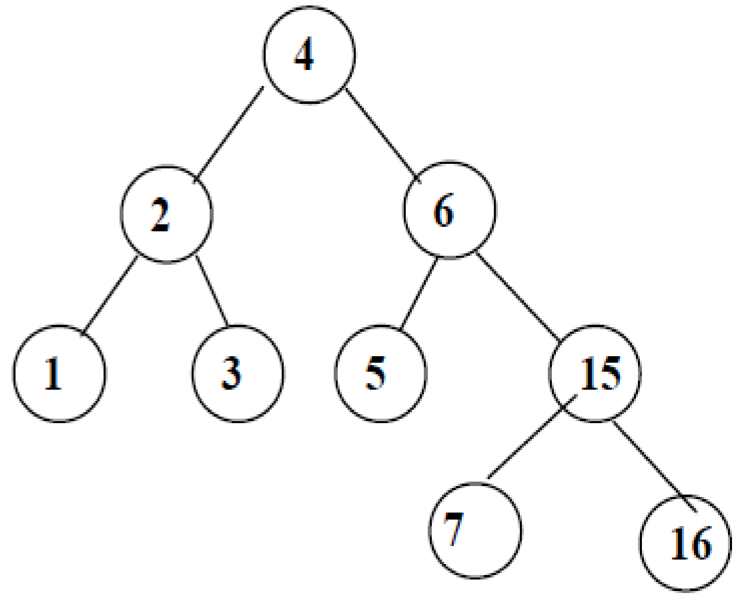


AVL Trees Example

Step 1: Rotate child and grandchild



Step 2: Rotate node and new child (AVL

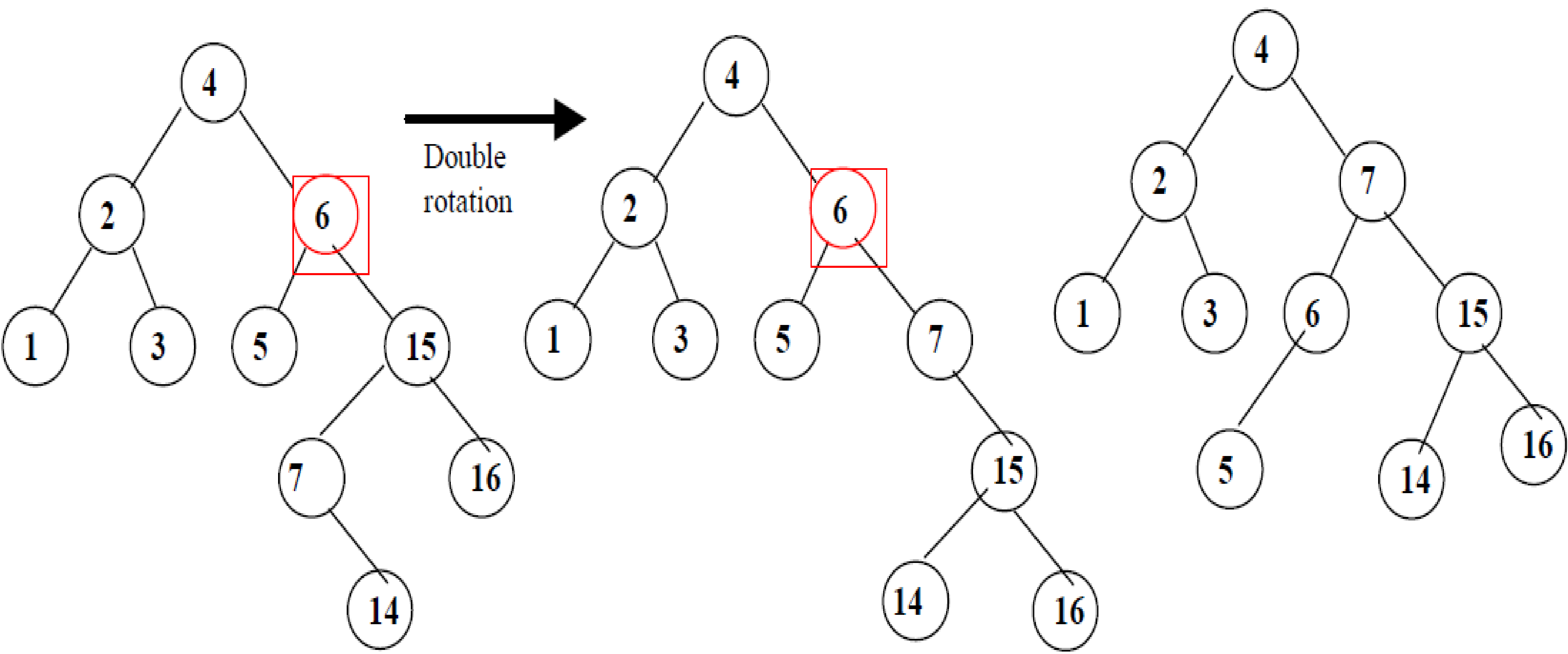


AVL Trees Example

Insert 14 (non-AVL)

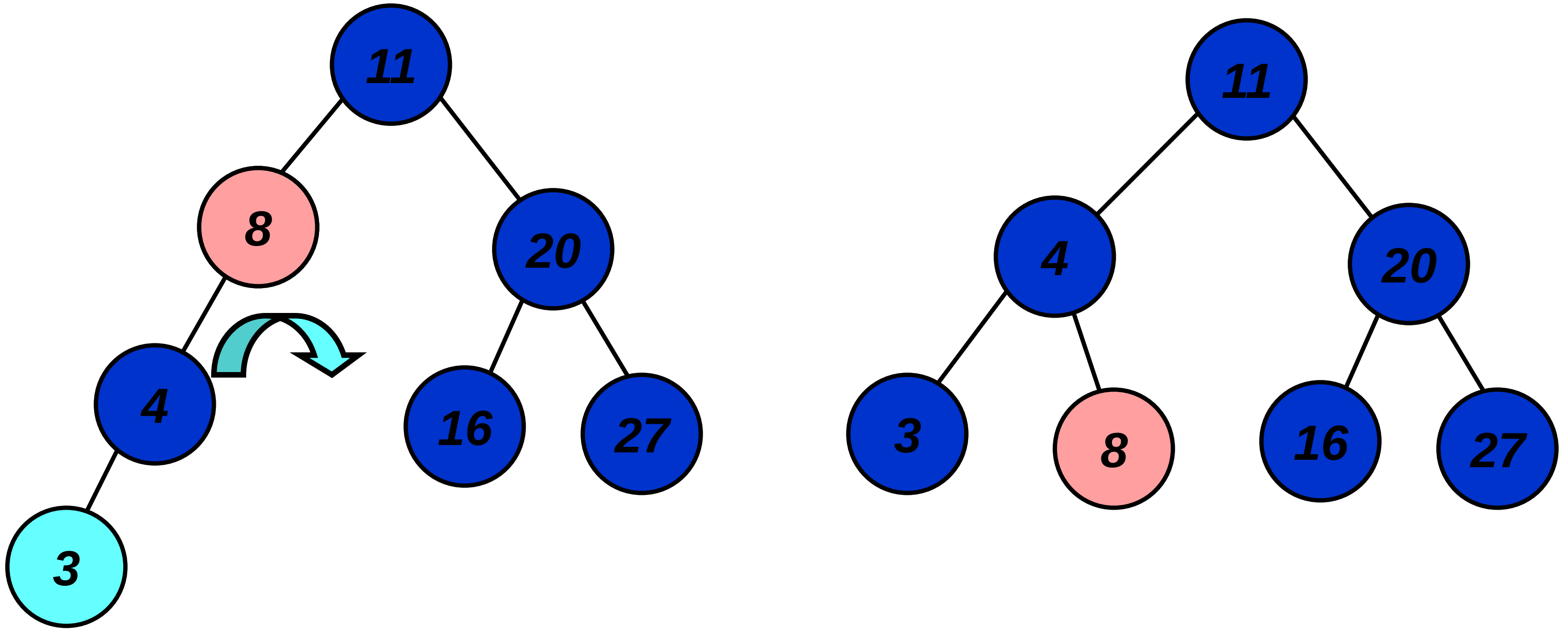
Step 1: Rotate child and grandchild

Step 2: Rotate node and new child (AVL)



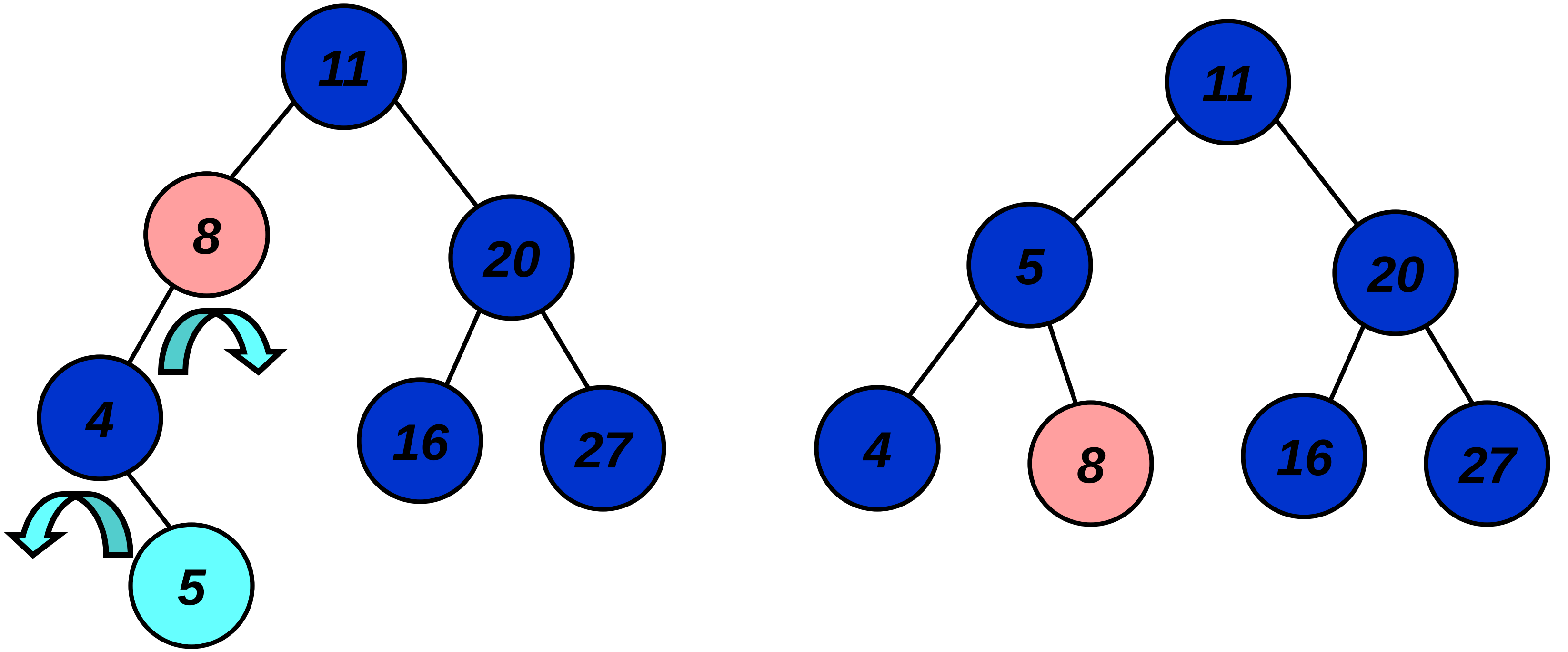
Example

- Insert 3 into the AVL tree



Example

- Insert 5 into the AVL tree



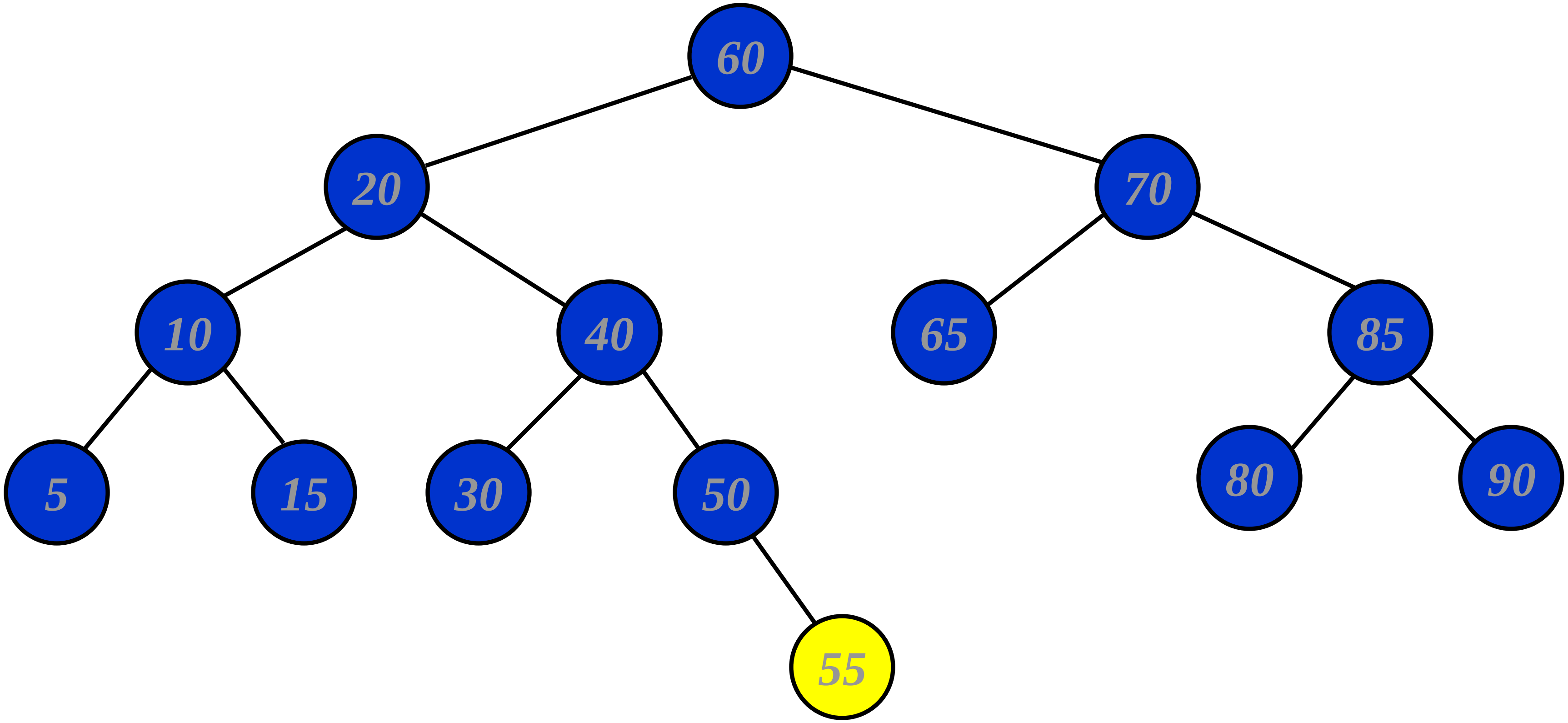
AVL Trees: Exercise

- Insertion order:
  - 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55

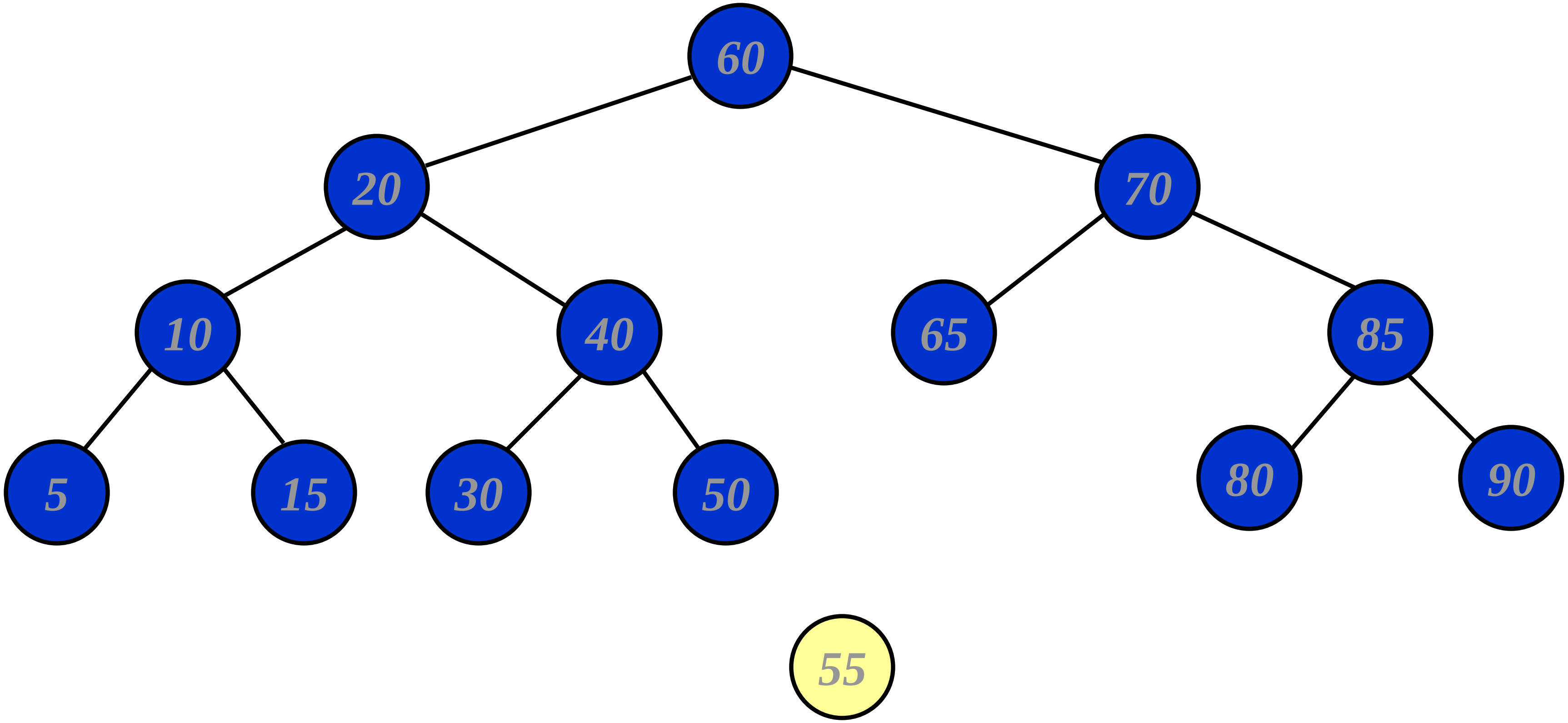
Deletion X in AVL Trees

- Deletion:
  - Case 1: if X is a leaf, delete X
  - Case 2: if X has 1 child, use it to replace X
  - Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)
- Rebalancing

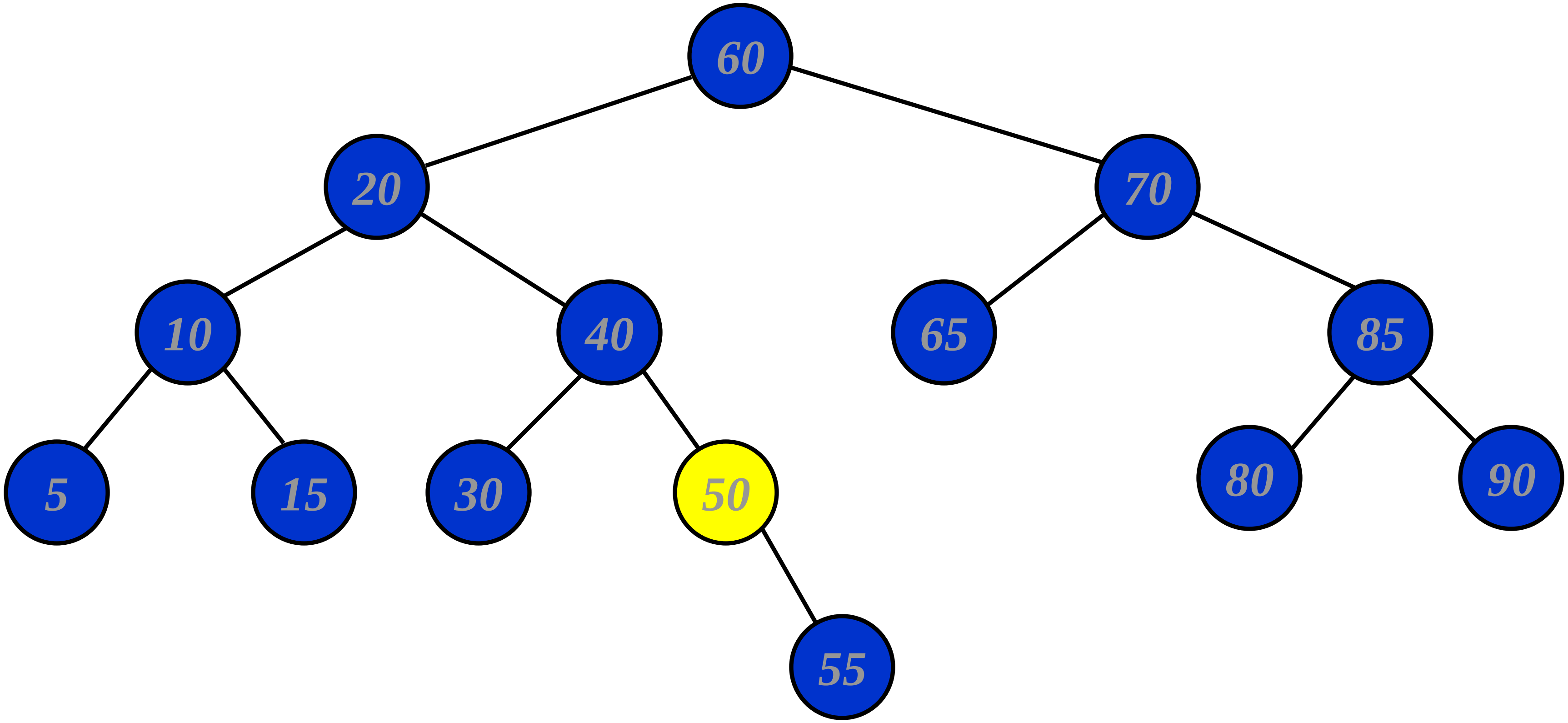
Delete 55 (case 1)



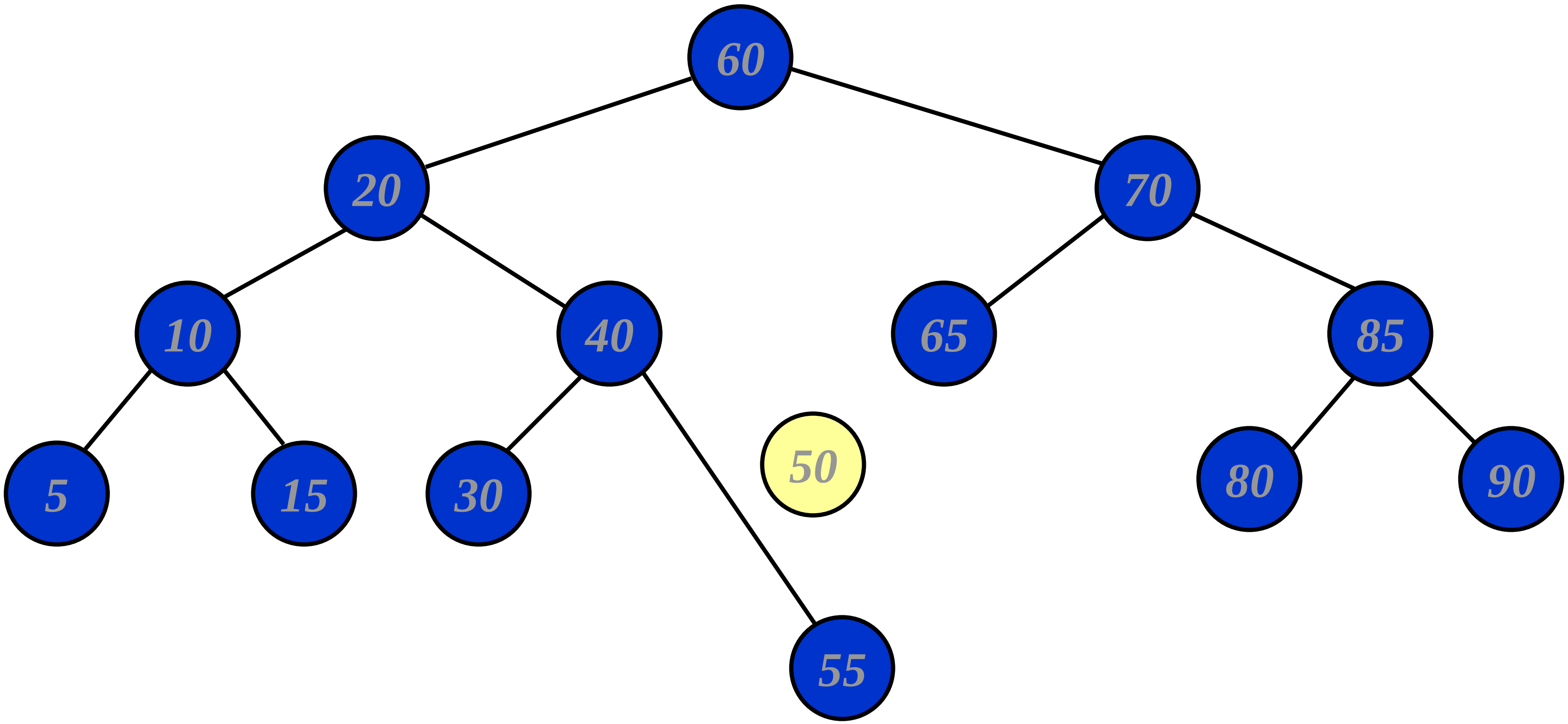
Delete 55 (case 1)



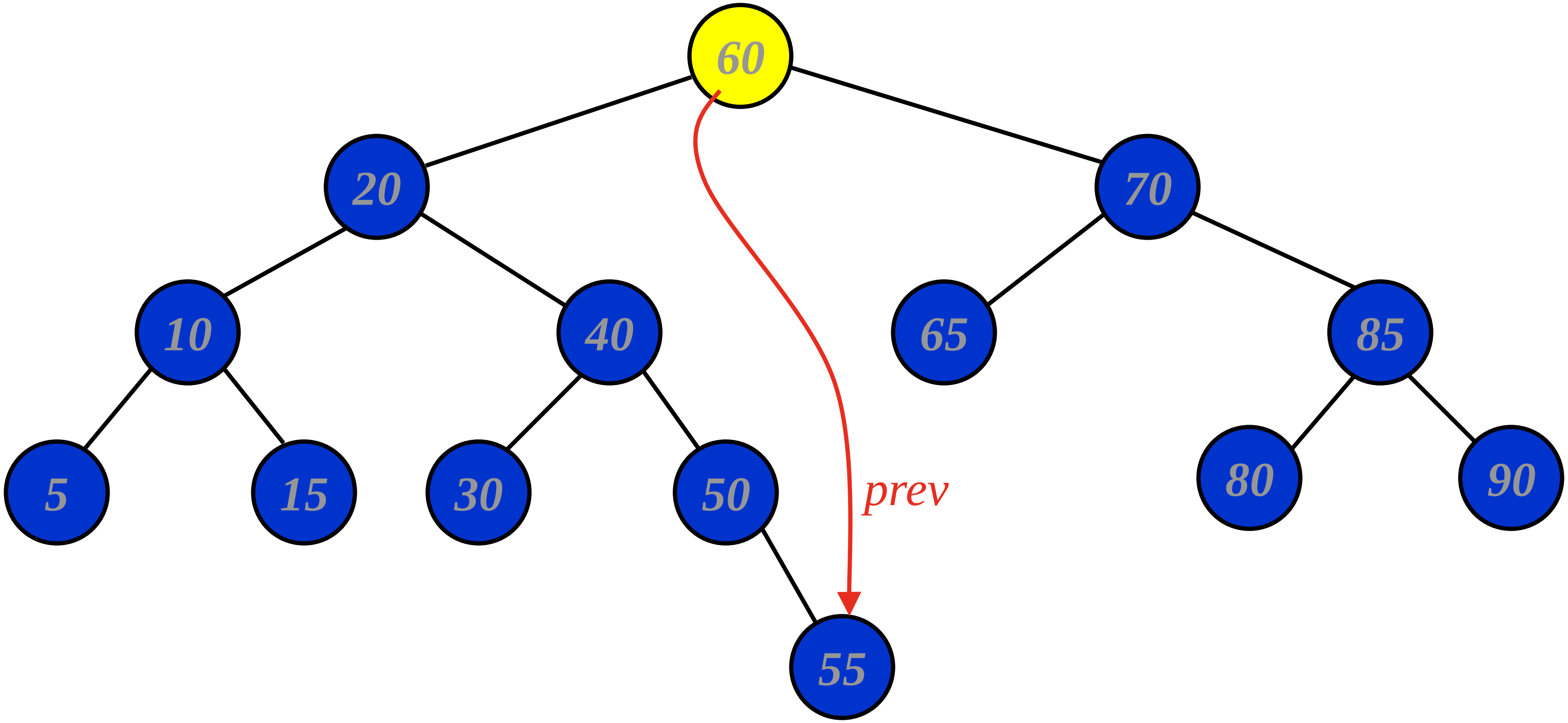
Delete 50 (case 2)



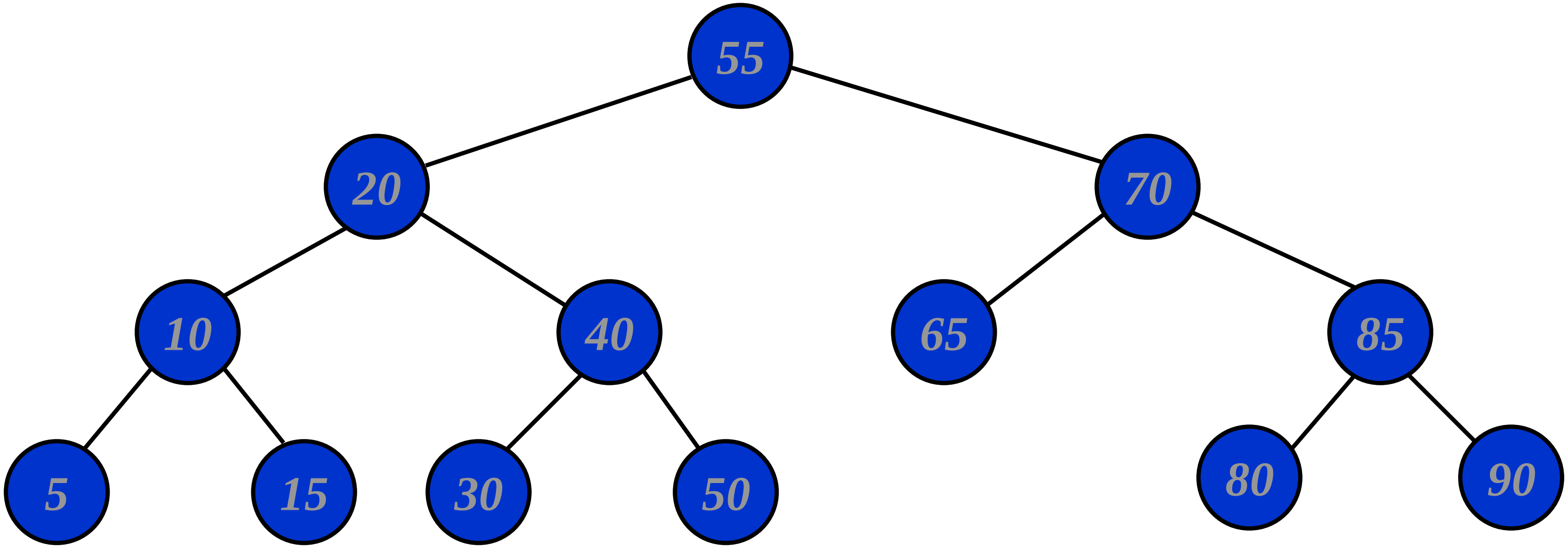
Delete 50 (case 2)



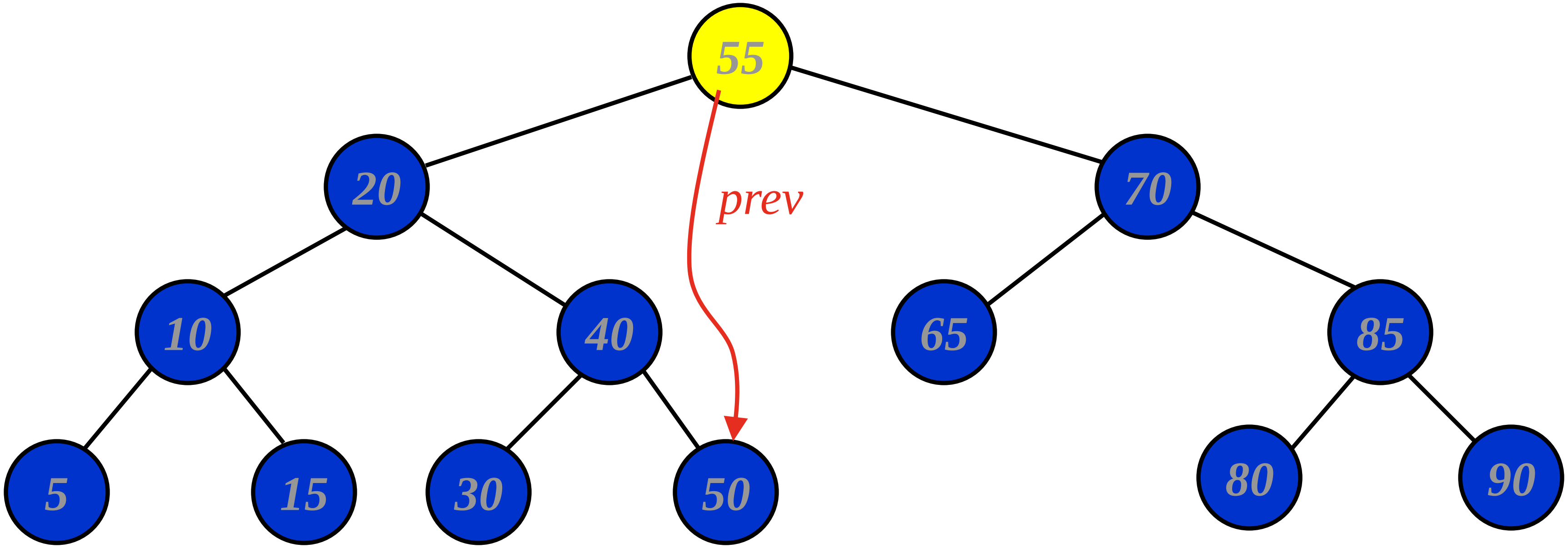
Delete 60 (case 3)



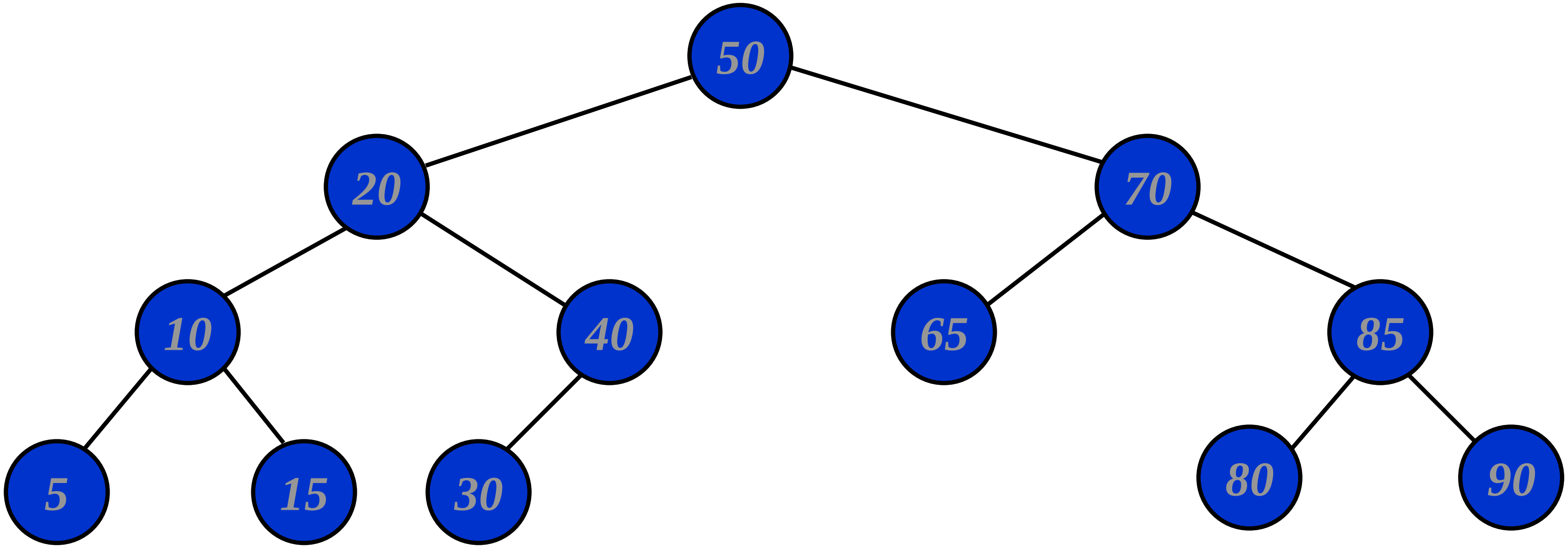
Delete 60 (case 3)



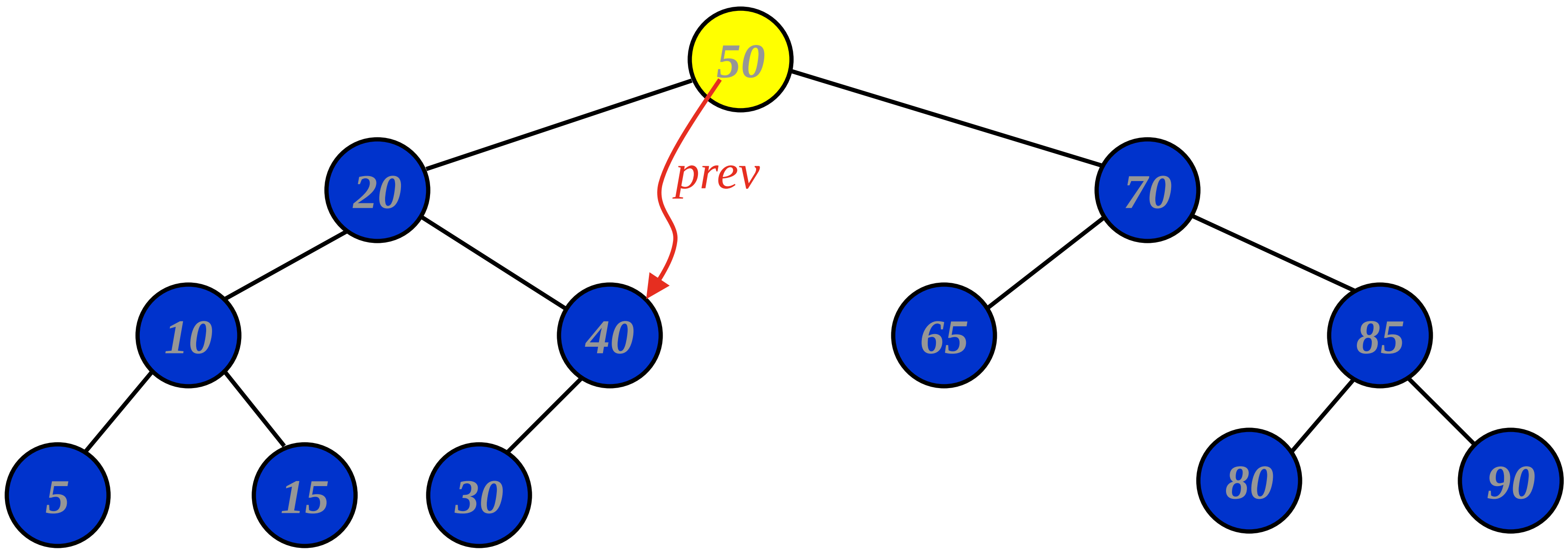
Delete 55 (case 3)



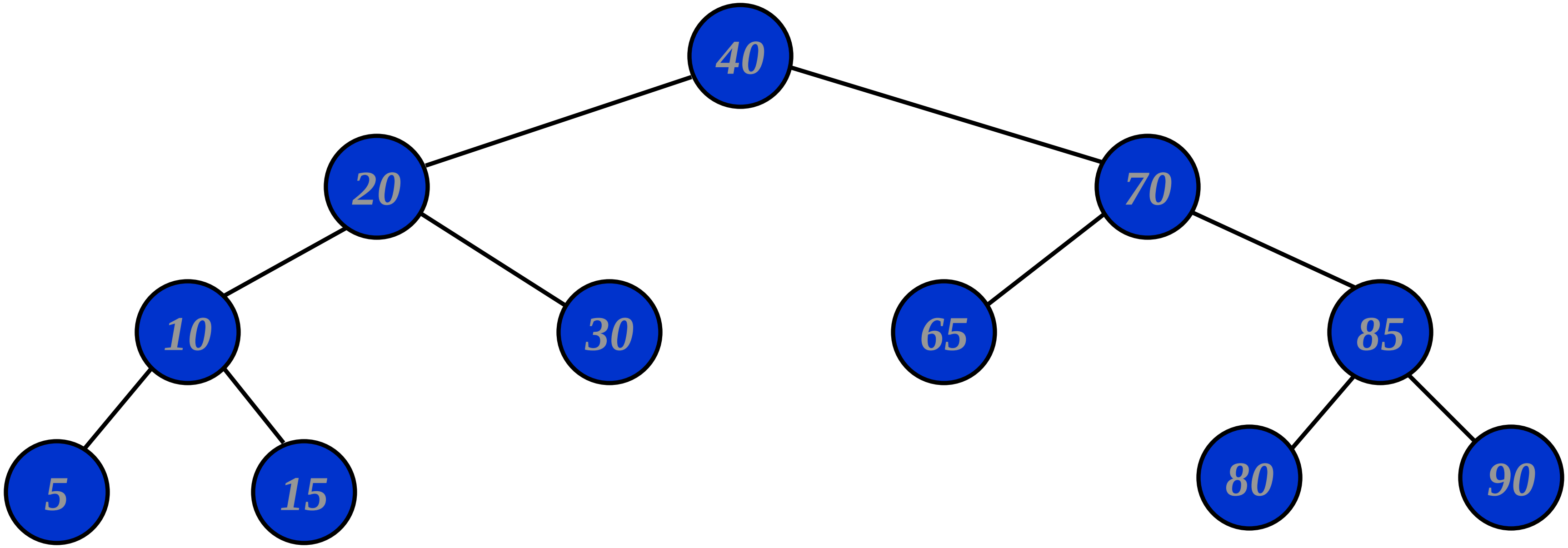
Delete 55 (case 3)



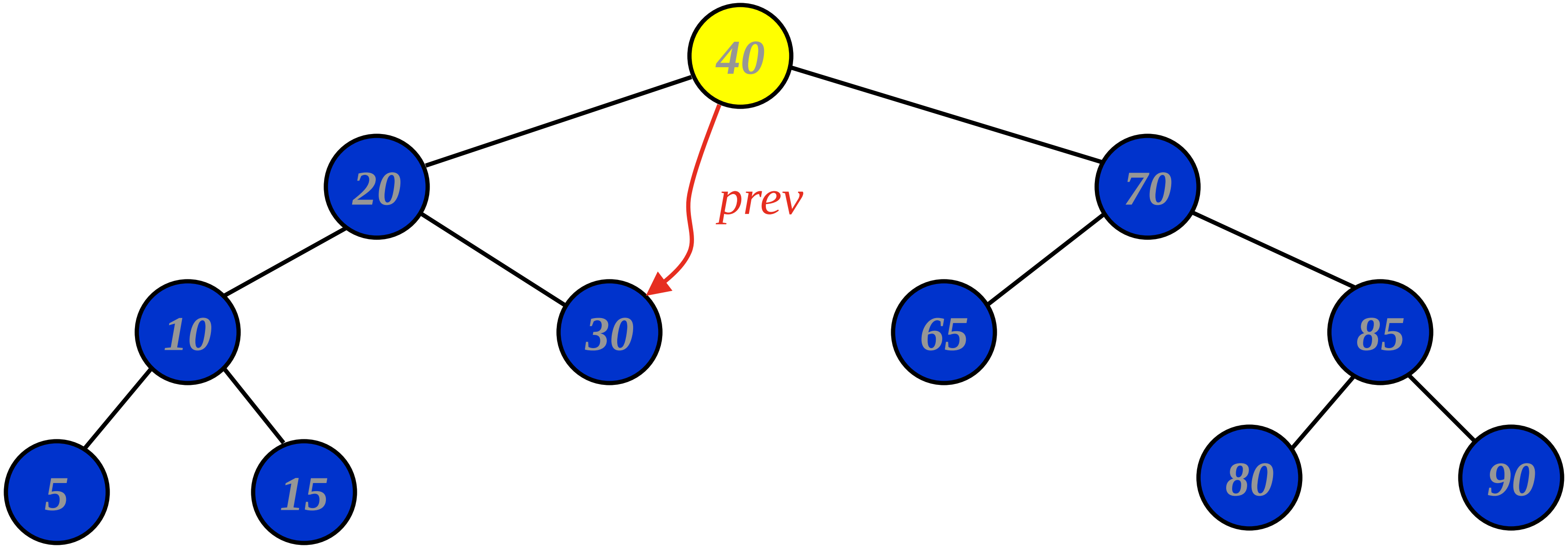
Delete 50 (case 3)



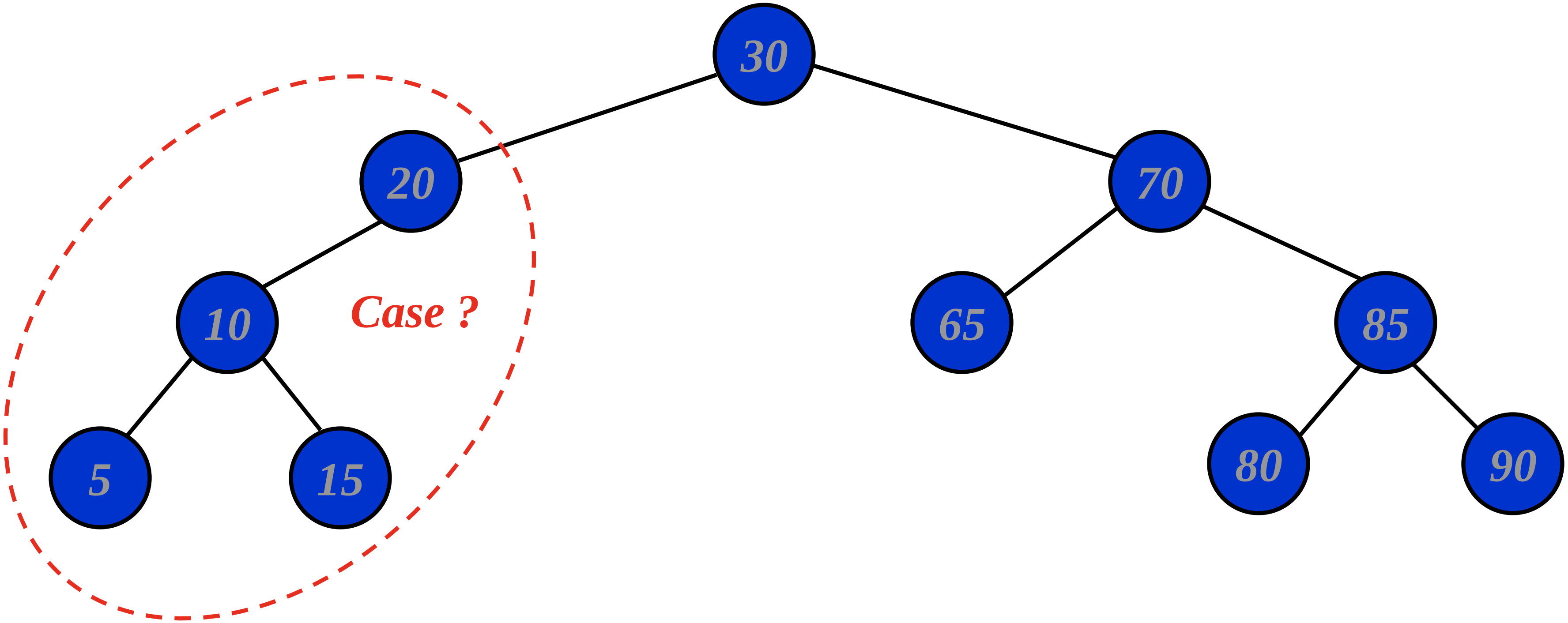
Delete 50 (case 3)



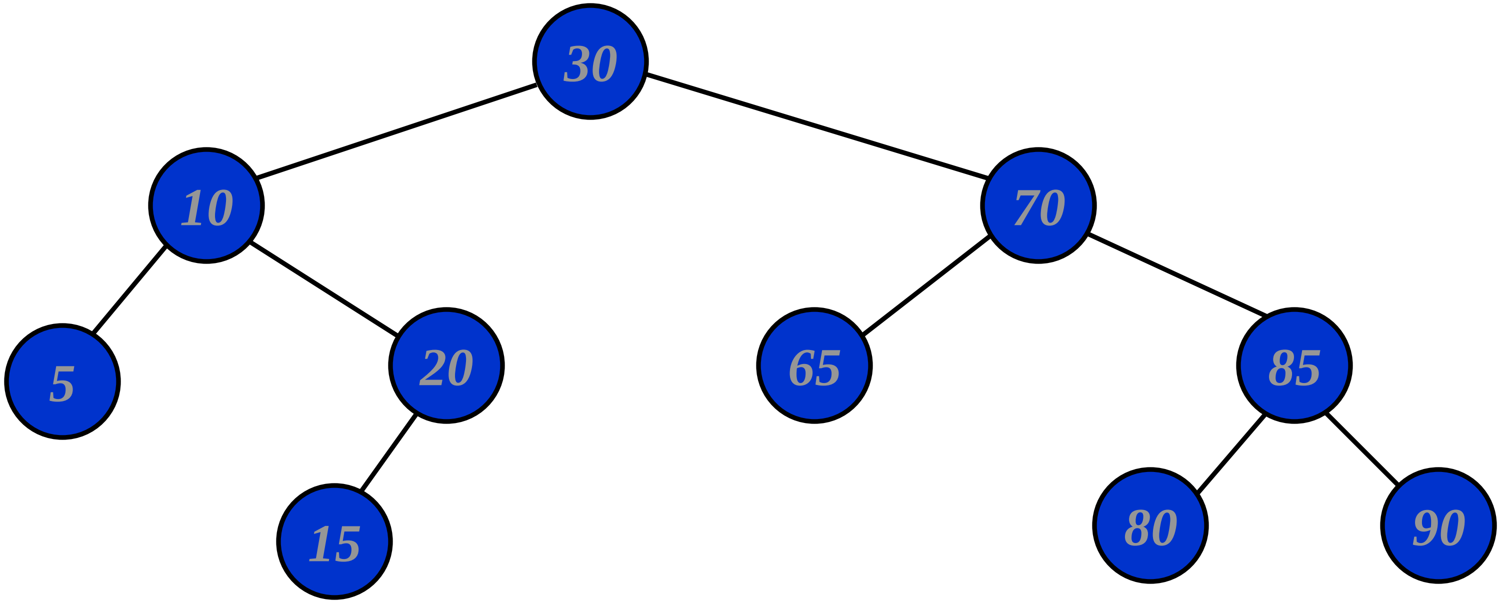
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



Single rotation is preferred!

AVL Tree: analysis

- The depth of AVL Trees is at most logarithmic.
- So, all of the operations on AVL trees are also logarithmic.
- The worst-case height is at most 44 percent more than the minimum possible for binary trees.